Approfondimenti tematici

PuntoInformatico Internet dal 1996 | Libri

iPhone PROGRAMMING

Il corso completo per imparare facilmente a programmare lo smartphone Apple

EDIZIONI MASTER

iPhone programming

Questo approfondimento tematico è pensato per chi vuol imparare a programmare e creare software per l'Apple iPhone.
La prima parte del testo guida il lettore alla conoscenza degli strumenti necessari per sviluppare sulla piattaforma mobile di Cupertino.
Le sezioni successive sono pensate per un apprendimento pratico basato su esempi di progetto: la creazione di un browser su misura, la gestione dell'interfaccia, la programmazione di un'agenda e di una to do list, la gestione corretta di celle e tabelle, l'utilizzo dell'accelerometro, la progettazione di un RSS reader e via dicendo.
Una serie di esempi pratici da seguire passo passo che – creando applicazioni testabili e perfettamente funzionanti - spingono il lettore a sperimentare sul campo il proprio livello di apprendimento e lo invitano a imparare divertendosi.

PuntoInformatico Internet dal 1996 | Libri

CREARE SOFTWARE PER L'APPLE IPHONE

INIZIA DA QUESTO NUMERO UN CORSO DEDICATO ALLO SVILUPPO DELL'APPLE IPHONE, LO SMARTPHONE CHE HA RIVOLUZIONATO IL MONDO MOBILE. IN QUESTO PRIMO APPUNTAMENTO FAREMO LA CONOSCENZA DI TUTTI GLI STRUMENTI NECESSARI

A causa della NDA (*Non Disclosure Agreement*) che nel mese di luglio dello scrso anno ha accompagnato la distribuzione dell'SDK, parallelamente all'uscita in Italia dell'iPhone, non ci è stato possibile pubblicare i successivi articoli pianificati per il seguente corso di programmazione di questo interessante dispositivo. Negli ultimi mesi sono state apportate numerose migliorie al firmware (terminate con il rilascio della versione stabile 2.2.1), e recentemente l'OS 3.0 con il relativo SDK 3.0, (in fase di beta 3 al momento della scrittura di questo articolo), fornirà ancora più strumenti per i programmatori che desiderano utilizzare al meglio le funzionalità di questa piattaforma di sviluppo. Molti concetti fondamentali, tra cui architettura dell'iPhone, su come è strutturato il framework, la disposizione delle cartelle dei progetti creati con Xcode, inclusi numerosi consigli di ottimizzazione, sono stati già affrontati nel numero 130 di questa rivista. Consigliamo a questo proposito di ridare una lettura a tali pagine, perché non verranno trattati, se non quando sarà strettamente necessario. Ricordiamo, inoltre, che è possibile realizzare un software completamente funzionante senza avere acquistato alcun iPhone o iPod Touch di seconda generazione, e senza acquistare la licenza. Di contro, si devono accettare le seguenti limitazioni: in alcune situazioni tale software, eseguito all'interno dell'emulatore, non si comporterà in maniera fedele al vero dispositivo, quello fisico, rendendo addirittura inutilizzabile il vostro prodotto. Questo è dovuto sia al fatto che l'emulatore suddetto viene eseguito utilizzando le risorse hardware (nettamente più prestanti sotto ogni aspetto) del vostro computer, sia perché manca di alcune funzionalità, come il GPS e l'accelerometro ad esempio, sia per alcune discrepanze che molti programmatori hanno riscontrato nei vari test; se non acquisterete una licenza non potrete testarlo su alcun iPhone/iPod e non potrete venderlo sull'Apple Store. La scelta se utilizzare un iPhone o un iPod Touch spetta a voi, un iPod Touch di seconda generazione è comunque valido, poiché ha un hardware leggermente più prestante rispetto al "fratello", manca però delle funzionalità GPS, UMTS e fotocamera; come detto precedentemente lo sblocco del Bluetooth con il firmware 3.0 nell'iPod Touch ha eliminato questa ulteriore differenza tra i due dispositivi.

REGISTRAZIONE E SDK

Per iniziare a programmare è necessario effettuare due operazioni: la prima consiste nel registrarsi come sviluppatore, divenire quindi *Registered iPhone Developer*, operazione completamente gratuita. Basterà accedere al seguente indirizzo http://developer.apple.com/iphone/ cliccare poi sulla voce in alto, *register*, e decidere se creare un nuovo account, oppure, nel caso vi siate già registrati utilizzando *mobileMe*, l'*itunes Store*, l'*ADC* (*Apple Developer Connection*) o *Apple Online Store*, utilizzare il vostro identificativo corrente. A questo punto è necessario inserire tutte le informazioni anagrafiche, accettare

Fig. 1: La home page del sottosito dedicato allo sviluppo di applicazioni per iPhone

REQUISITI

Conoscenze richieste
OOP

Software
Mac OS X 10.5 o superiore

Impegno

Tempo di realizzazione

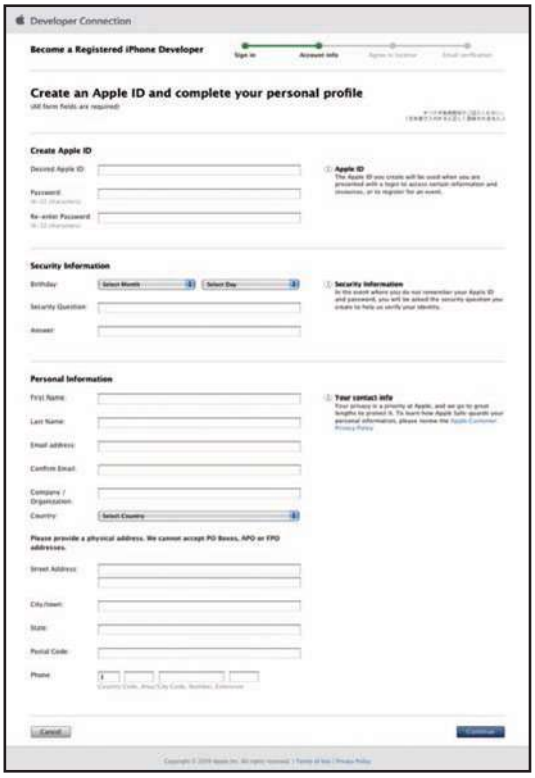

Fig. 2: La fase di registrazione, in cui sono richieste tutte le vostre informazioni anagrafiche

Fig. 3: La struttura della cartella di installazione dell'SDK

l'informativa e, alla ricezione dell'email di conferma, seguire il link che vi verrà inviato: siete così divenuti *Registered iPhone Developer*.
La seconda, ed ultima, operazione da effettuare, consiste nello scaricare e installare sul vostro Mac l'SDK; la versione stabile, 2.2.1, consiste in un file di circa 1.7GB, consigliamo quindi di utilizzare una connessione relativamente veloce; è inoltre necessario avere almeno Leopard aggiornato alla versione 10.5.4 e disporre di sufficiente spazio sul proprio disco, poiché, inclusa la documentazione, verranno occupati oltre 5GB. All'avvio dell'installer sarà richiesto quali componenti installare: quelli di default sono sufficienti. Al termine di questo non breve processo le librerie, tutto il software necessario, tra cui il simulatore, Xcode, l'editor per i progetti, Interface Builder, l'editor WYSIWYG, Shark e Instruments, necessari per il tuning delle vostre applicazioni, compresa una versione parziale della documentazione saranno contenuti all'interno della cartella *Developer*. All'interno della cartella *Applications* trovano posto quegli strumenti che adopererete in ogni progetto: XCode, Interface Builder, Shark (sottocartella *Performance Tools*) e Instruments; poiché da Xcode è possibile richiamare tutti gli altri applicativi, basterà trascinare solo tale software nel Dock per avere disponibile tutti questi programmi con un colpo di clic. La documentazione non viene installata completamente, e si ha libera scelta se consultarla richiamando le pagine disponibili online, prelevate automaticamente quando necessario, risparmiando spazio sul proprio HD, o scaricarla localmente; in questo ultimo caso, sarà necessario avviare XCode, selezionare il menu *Help->Documentation* e cliccare sui singoli tasti *get* che si trovano vicino alle singole categorie; sono inoltre disponibili documenti riguardanti la libreria *WebObjects* (bisogna selezionarla tra i componenti aggiuntivi durante la fase di installazione) e le API Java 1.4 e 1.5, questo perché la versione di Xcode (e degli altri software installati) consente anche di realizzare applicativi, plugin, moduli aggiuntivi per Mac OS in Java, C++, Ruby e altri linguaggi. La versione 2.2.1 non è dotata di documentazione, poiché è stata rilasciata per correggere alcuni bug, non è quindi avvenuta alcuna modifica all'API, per tale motivo la documentazione più aggiornata corrisponde a quella fornita con la 2.2. Utilizzando il campo di testo presente in alto a destra, si ricercheranno quelle funzioni e/o metodi disponibili per una consultazione; è possibile inoltre filtrare su quale API effettuare la ricerca.

INIZIAMO L'AVVENTURA

In questa prima puntata creeremo un browser web minimale che ci permetterà di conoscere i concetti fondamentali di questa tecnologia, progressivamente presenteremo le nozioni necessarie per completare il progetto e renderlo pienamente operativo; non creeremo nuove classi per evitare confusione, ma utilizzeremo quelle fornite quando si crea un nuovo software usando il wizard; ricordiamo che la struttura delle classi con cui lavoreremo è organizzata con una struttura ad albero (Fig.2), avente una radice (*NSObject* al più alto livello, *UIWindow* scendendo di qualche gradino) a cui si associano i vari

RIFERIMENTI WEB

Per la creazione dell'account e per scaricare l'SDK, visitate il seguente percorso web: http://developer.apple.com/iphone/

All'indirizzo che segue trovate una descrizione dettagliata delle novità fornite con il nuovo SDK 3.0: http://developer.apple.com/iphone/prerelease/library/releasenotes/General/WhatsNewIniPhoneOS/Articles/iPhoneOSv3.html

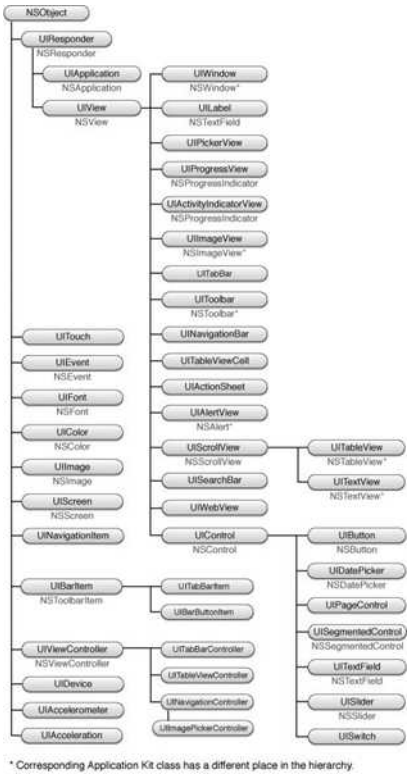

Fig. 4: La struttura ad albero dell'API

UIView Controller (contenitori invisibili *UIVIEW*) e i cui figli sono discendenti della classe *UIView* (bottoni, testi e altri componenti).
L'approccio migliore per iniziare a programmare l'iPhone/iPod Touch è quello top-down, adoperando quei componenti dell'interfaccia grafica predefiniti, disponibili attraverso l'Interface Builder (IB), aggiungendo, quando necessario, il codice minimale richiesto per ottenere il risultato desiderato. Successivamente si potrà, nel caso lo si trovasse limitante oppure per semplici gusti personali, abbandonare completamente tale strumento visuale, e realizzare tutto in puro Objective-C. Riprendiamo brevemente il discorso, iniziato nell'articolo di presentazione sull'iPhone, pubblicato a settembre e riguardante il design pattern MVC; dovrete metabolizzare tale concetto per non riscontrare problemi nella fase di sviluppo del vostro primo progetto. MVC è acronimo di *Model View Controller*, indica i tre componenti utilizzati per realizzare applicazioni in grado di adattarsi meglio a variazioni della propria struttura interna; con *Model* si indica la/le struttura/e dati utilizzata/e per gestire le informazioni necessarie per un corretto funzionamento del vostro software, possono essere variabili, array, strutture più complesse, file locali o remoti e anche database; con *View* (che non a caso viene identificata con la classe *UIView*) si indica l'interfaccia grafica necessaria per consentire all'utente finale una consultazione e/o interazione con tali dati; lo strumento per realizzarle è *Interface Builder* (utilizzando i file *xib*) oppure puro codice Objectice-C (creando tutte le strutture grafiche all'interno di file *.m/.h*); con *Controller* intendiamo quella parte di codice necessaria per una corretta comunicazione tra le due strutture precedenti: è, di fatto, la componente in cui risiede "l'intelligenza" del vostro software (e dove ovviamente riscontrerete la quasi totalità dei problemi e degli errori), provvedendo a creare un mapping tra cosa devono mostrate i vari componenti grafici e in che modo devono essere modificati conseguentemente a un'interazione da parte dell'utilizzatore del vostro software; in questo contesto il Controller (anche qui non a caso identificato con la classe *UIViewController*) corrisponde al codice presente nei file *.m/.h*, realizzati con il linguaggio Objective-C (è possibile comunque inserire codice C e C++ nel caso fosse necessario velocizzare alcune operazioni o per utilizzare le API a livello più basso).

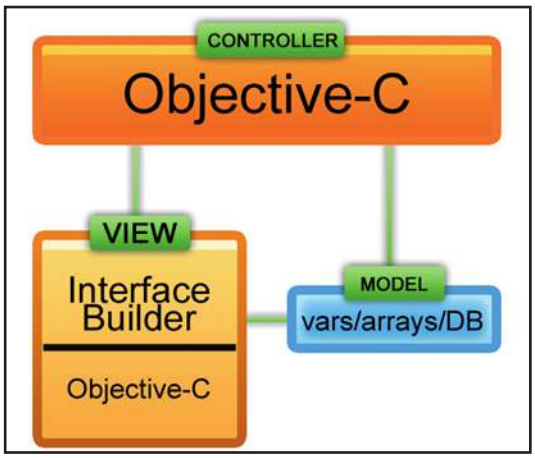

Fig. 5: Una generalizzazione del design pattern MVC utilizzato

CREAZIONE DEL PROGETTO E XCODE

Il progetto Xcode fornito dal Wizard, che ci consente di realizzare il nostro browser nella maniera più veloce, utilizzando un'istanza di un *UIViewController* contenente una della classe *UIView* si chiama *View-Based Application*.
Decidiamo il nome dell'applicativo, e posizioniamolo in una cartella a nostra scelta. Al termine della creazione ci verrà presentata la schermata di Fig.7. Analizziamola in breve: è suddivisa principalmente in tre aree: nella parte alta troviamo i comandi necessari compilare il nostro software, selezionare la piattaforma di testing, simulatore o dispositivo fisico (nel caso abbiate già acquistato la licenza), nella colonna di sinistra trovia-

NOTA

DESIGN PATTERN MVC

Model-View-Controller (MVC, talvolta tradotto in italiano Modello-Vista-Controllore) è un pattern architetturale molto diffuso nello sviluppo di interfacce grafiche di sistemi software object-oriented. Viene sovente utilizzato da framework basati su Ruby, Java (Swing, JSF e Struts), su Objective C o su .NET. Il pattern è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali:

- il model fornisce i metodi per accedere ai dati utili all'applicazione;

- il view visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti;

- il controller riceve i comandi dell'utente (in genere attraverso il view) e li attua modificando lo stato degli altri due componenti

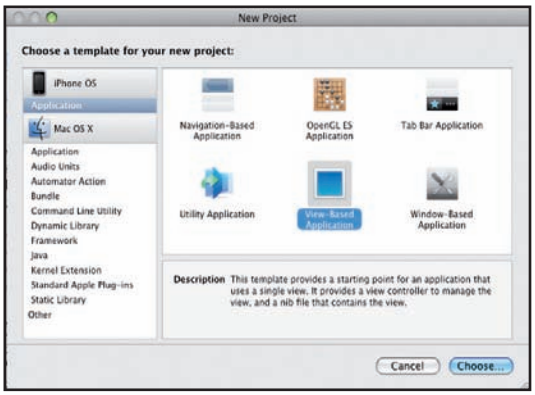

Fig. 6: La finestra di scelta del progetto

mo tutti i file del progetto, comprese le librerie utilizzate, in quella destra viene visualizzato il contenuto del file selezionato a sinistra.
Per aggiungere un file al nostro progetto è sufficiente crearlo utilizzando il menu *File->Nuovo File*, oppure ne trasciniamo uno preesistente all'interno della colonna di sinistra, prelevandolo da una finestra di Finder; è possibile utilizzare anche il comando *Add to Project* presente all'interno del menu *Project*, presenterà una finestra da cui selezionare uno o più file presenti nel proprio computer: questa operazione è necessaria quando si creano risorse con software diversi come audio, video o immagini. Copiare tramite Finder un file all'interno della cartella del nostro progetto non è sufficiente, poiché XCode richiede che i file che dovrà compilare vengano aggiunti utilizzando una delle modalità suddette: è quindi possibile inserire qualunque tipo di file nella cartella del progetto e aggiungere solo un sottoinsieme di tali elementi in XCode. Dopo aver selezionato il file ci verrà presentata una finestra in cui potremo selezionare il checkbox corrispondente alla voce *Copy items into destination group's folder (if needed)*, tale operazione non è obbligatoria, ma in caso non la si attivasse non si copierebbe localmente, all'interno della cartella del progetto e si lavorerebbe con un link simbolico; una cancellazione o uno spostamento del file originale (quello fisico) dalla sua posizione (è situato in una diversa cartella del proprio computer) non permetterebbe di compilare il progetto; selezionando quindi tale opzione, abbiamo la garanzia di avere all'interno della cartella del progetto tutti i file necessari, rendendo estremamente veloce qualunque operazione di backup o invio ad altri sviluppatori. Analizziamo ora la finestra di sinistra. Noterete alcune cartelle, di colore giallo, queste non esistono fisicamente nel vostro computer, si chiamano *Group(s)* e hanno lo scopo di consentire un raggruppamento visivo delle proprie risorse, sono delle cartelle virtuali, è possibile quindi crearle liberamente utilizzando il tasto destro del mouse *(Add->New Group)* o dal menu *Project*.

La cartella *frameworks* contiene le librerie che si desidera inserire nel progetto, le tre fondamentali, inserite automaticamente, sono *UIKit.framework*, *Fondation.framework e Core Graphics.framework*; per aggiungerne altre basterà trascinarle all'interno del progetto, oppure utilizzare il destro del mouse e selezionare *Add Existing Frameworks*. Le altre cartelle virtuali, rappresentate con differenti icone, svolgono principalmente funzione di raggruppamento automatico di diversi tipi di informazioni, errori di programmazione, file grafici xib, bookmark, ad esempio. Premendo *CTRL + INVIO*, oppure selezionando l'icona *Build & Go*, o dal menu *Build* la voce *Build & Run* avvieremo il simulatore.

I FILE XIB E L'INTERFACE BUILDER

Ora che abbiamo creato un progetto, iniziamo a descrivere i due componenti fondamentali che sono onnipresenti nei progetti per iPhone, in maniera diretta e/o indiretta (utilizzando componenti che da questi derivano): *UIViewController* e *UIView*. Ogni istanza di UIViewController è invisibile, e il suo scopo è quello di contenitore delle UIView (a cui appartengono tutti i componenti visuali, bottoni, testi inclusi) che in esso andremo a inserire, può svolgere anche ruolo di gestore degli eventi, rafforzando ulteriormente il concetto di come nell'MVC il Controller sia "trasparente", non abbia quindi un aspetto grafico, ma sia puramente codice. Dopo avere creato il progetto troveremo due file *xib* all'interno della colonna di sinistra di XCode, un primo chiamato *MainWindow.xib*, il quale contiene la *UIWindow*, la root del nostro software, che provvede automaticamente a contenere e gestire tutte le sottofinestre (e la catena di eventi), nel nostro caso sono un viewcontroller contenente una view, e un secondo file il cui nome, terminante per *ViewController*, dipenderà da quel-

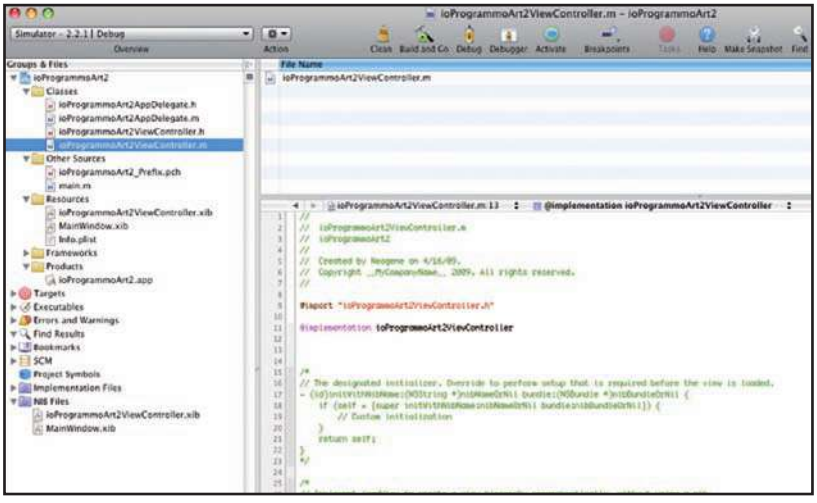

Fig. 7: L'interfaccia di sviluppo proposta dal tool XCode

NOTA

INTERFACE BUILDER

Interface Builder è un'applicazione facente parte di Xcode. Consente agli sviluppatori che usano Carbon e Cocoa di disegnare interfacce grafiche per le applicazioni usando uno strumento grafico, senza la neccessità scrivere nessuna riga di codice. L'interfaccia risultante è salvata in un file *.nib* (abbreviazione di *NeXT Interface Builder*).

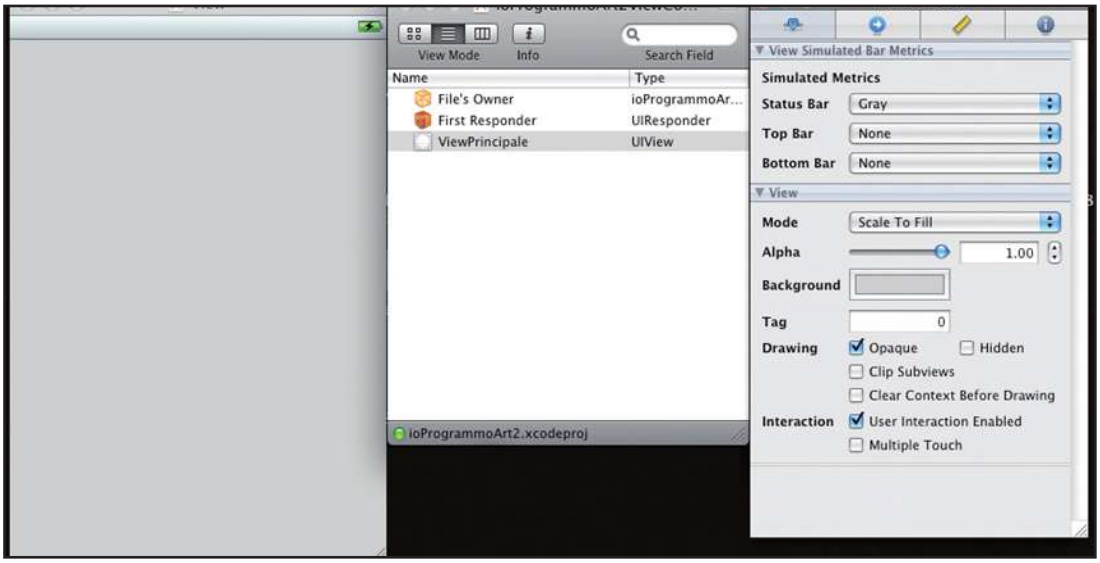

Fig. 8: L'interfaccia di Interface Builder, il wizard che consente di "disegnare" la nostra applicazione

NOTA

APPLE SDK IPHONE 3.0

iPhone OS 3.0, la cui release ufficiale dovrebbe essere rilasciata in questi giorni, include un Software Developer Kit (SDK) aggiornato con oltre 1000 nuove Application Programming Interface (API) tra cui In-App Purchases, connessioni Peer-to-Peer, una interfaccia applicativa per gli accessori, accesso alla libreria musicale dell'iPod, un nuovo Map API e le notifiche Push. Saranno disponibili oltre 100 nuove funzioni per gli utilizzatori di iPhone e iPod touch, tra cui il Taglia, Copia e Incolla (che potranno essere utilizzate all'interno e tra le applicazioni), gli MMS (disponibili solo su iPhone 3G) per spedire e ricevere immagini, contatti, file audio e posizioni geografiche con la funzione Messaggi, il Bluetooth stereo, la sincronizzazione delle note con Mac e PC; shake per attivare la funzione Shuffle, il controllo parentale per i programmi televisivi, i film e le applicazioni dell'App Store e il login automatico agli hot spot Wi-Fi.

lo del nostro progetto, nel nostro caso è *ioProgrammoArt2Viewcontroller.xib*; quest'ultimo file è quello che ci interessa e che andremo a modificare visivamente con Interface Builder.
La nostra UIView, al cui interno andremo a posizionare alcuni componenti visuali (pulsanti, testi, immagini), ha necessità di essere ospitata in un UIViewController (o classe derivata) per essere mostrata su schermo, in questo caso tale operazione è stata gestita automaticamente dal wizard, ma in altri casi è necessario creare tramite IB tale relazione, oppure adoperando Objective-C. In un progetto sviluppato con XCode è possibile creare un numero indeterminato di file *xib*, realizzati adoperando Interface Builder, l'editor WYSIWYG disponibile tra i software forniti nell'SDK; ogni file con estensione *xib* è in realtà un semplice XML (per chi avesse utilizzato Adobe Flex è un approccio simile a quello dei file MXML, mentre per chi conosce Windows Presentation Foundation è simile, sempre concettualmente parlando, a quello ideato per i file XAML) che viene letto, interpretato e mostrato in maniera visuale da IB. Diamo uno sguardo veloce alla struttura di uno di questi file, premendo il destro del mouse selezioniamo la voce "*Open as Souce Code File*", otterremo il seguente codice (qui ridotto per la sua estrema lunghezza):

```
<?xml version="1.0" encoding="UTF-8"?>
<archive type="com.apple.InterfaceBuilder3.
                        CocoaTouch.XIB" version="7.02">
<data>
<int key="IBDocument.SystemTarget">528</int>
<string
      key="IBDocument.SystemVersion">9E17</string>

<string key="IBDocument.InterfaceBuilder
                        Version">672</string>
<string
      key="IBDocument.AppKitVersion">949.33</string>
...
<object class="IBUIView" id="774585933">
...
<string key="NSFrameSize">{320, 460}</string>
<object class="NSColor" key="IBUIBackgroundColor">
..
<int key="NSColorSpace">1</int>
<bytes key="NSRGB">MC44MDQzNDc4
      MSAwLjM5MzQ0OTI4IDAuMjk4MDAzOTcAA</bytes>
</object>
...
<reference key="object" ref="774585933"/>
<reference key="parent" ref="360949347"/>
<string key="objectName">ViewPrincipale</string>
....
</data>
</archive>
```

Un file XML relativamente complesso, ma basta fare qualche test all'interno di IB per verificare in che modo questa struttura viene modificata; se notate, la riga *<object class="IBUIView" id="774585933">* contiene al suo interno molte delle informazioni riguardanti la view che è stata creata automaticamente, ad esempio le dimensioni, il colore di sfondo e quali componenti vengono simulati visivamente (spiegheremo più in avanti tale funzionalità); il numero ID viene usato in altre locazioni di tale file per riferimenti a questo stesso oggetto (come avviene in una delle ultime righe *<reference key="object" ref="774585933"/*.
Modificare direttamente tale file è inutile, poiché IB serve proprio a questo, inoltre si rischia di renderlo inutilizzabile, ma sapere almeno che è un formato leggibile da qualunque utente e non è codificato/compilato, può risultare utile in alcuni casi,

come quando si corrompe per qualche motivo (crash di IB, del sistema operativo, o della locazione fisica in cui viene memorizzato) e non si riesce più a modificare una (o più) proprietà di un (o più) componente da IB.
Questo formato viene automaticamente convertito in fase di compilazione in maniera trasparente. Cliccando invece due volte sul tale file avvieremo automaticamente IB.
Partendo da sinistra e andando verso destra abbiamo: la finestra di anteprima, la Document Window e l'Inspector; la prima è responsabile della visualizzazione dell'anteprima dell'interfaccia grafica, mostrerà quindi tutti i componenti grafici che inseriremo con una semplice operazione di trascinamento dalla finestra chiamata *Library*. La seconda finestra che incontriamo, la *Document Window*, mostra un elenco di tutti gli oggetti presenti nel file *.xib*, sia grafici che non, ci permette, inoltre, di selezionarli e poter così visualizzare i dettagli all'interno della finestra alla sua destra, l'Inspector; la chiusura della Document Window comporta la chiusura di tutto il file *xib* in IB, rendendo quindi necessario riaprire tale file dal menu *File->Open* o facendo nuovamente doppio clic in XCode. I tre componenti visibili nel Document Window inizialmente sono il file owner, il cui scopo è quello di identificare quale classe (la coppia *.m/.h*) è collegata al file *.xib* che stiamo usando (sotto la colonna *type* risulta *ioProgrammoArt2ViewController*, creata automaticamente dal wizard), *first responderer* utilizzato per gestire la catena di eventi, e la nostra View, che in questo caso non ha alcuna coppia di file *.m/.h* associata (basta notare che sotto la colonna *Type* risulta *UIView* invece di un nome di classe creato manualmente dal sottoscritto). Le quattro sottofinestre dell'Inspector forniscono tutte le informazioni riguardanti un determinato componente selezionato. Cliccando sulla *view,* utilizzando il *Document Window* abbiamo ottenuto i seguenti risultati. *Attributes Inspector*: nella parte alta troviamo la sezione chiamata *view simulated bar metrics*, che permette di aggiungere solo visivamente alcuni componenti grafici, tipici di classi derivate da *UIViewController* più complesse, come il *Tab Bar Controller* e il *Navigation Controller*, in modo da fornire un feeback visivo di come tale view cambierà le proprie proporzioni all'interno dei contenitori; l'anteprima non modifica in alcun modo la struttura effettiva della view, è un semplice strumento di valutazione. Nella parte bassa sono presenti opzioni come la trasparenza, il colore della view, come questa si adatta in funzione delle dimensioni settate, se consente l'interazione con l'utente, e altre funzioni specifiche a seconda della sottoclasse di *UIView* utilizzata (ad esempio, un UIButton, un semplice bottone, ha proprietà diverse di un *UILabel*, il cui scopo è solo di visualizzare

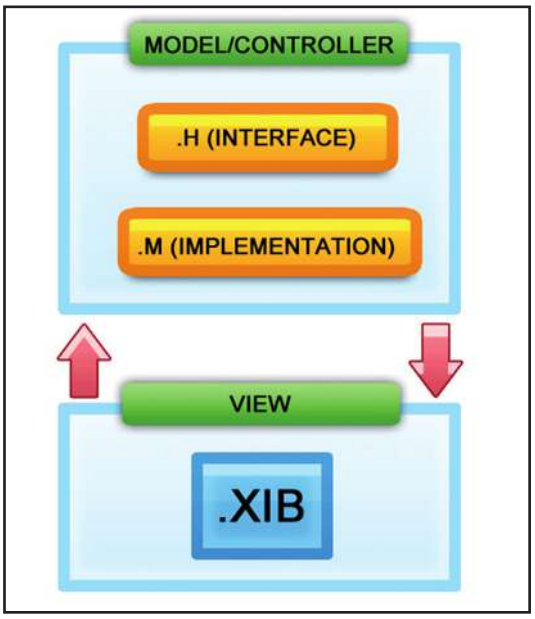

Fig. 9: Come si identificano i componenti dell'MVC utilizzando un file XIB

del testo); spiegheremo in dettaglio tutti questi campi nei prossimi articoli a seconda delle esigenze. Nel *Connection Inspector* avremo modo di visualizzare i collegamenti tra view e controller (ma non solo). Nella *Size Inspector* troviamo i tipici controlli di dimensionamento e allineamento.
Nell'Identity Inspector trovano posto tutte le informazioni riguardanti relazioni ed eventi gestiti dalla view, compresa la classe che la gestisce (si parlerà di *IBOutlet* e *IBAction*).
Come spiegato precedentemente, IB genera file che appartengono al componente del pattern MVC chiamato *View*, non c'è modo diretto di scrivere all'interno di questo editor codice Objective-C, per tale motivo dobbiamo creare una coppia di file per ogni componente grafico con il quale vogliamo che l'utente interagisca (sia derivato da *UIViewController* che da *UiView*), o che desideriamo animare o trasformare in qualche modo, e associare in Interface Builder questa coppia di file

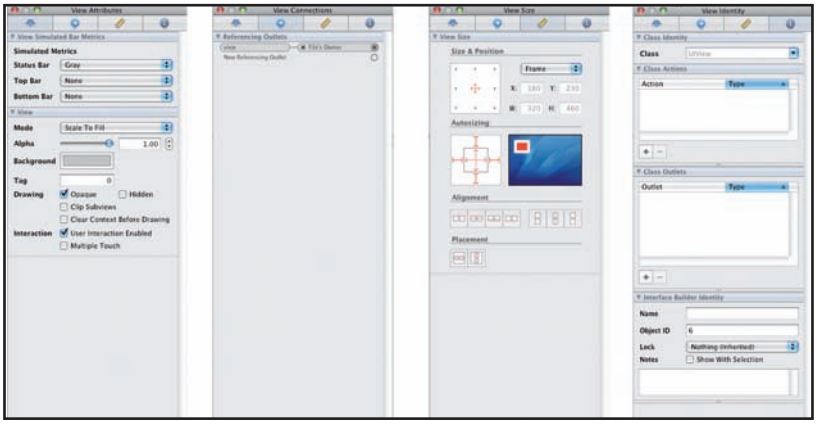

Fig. 10: L'Attributes Inspector dell'Interface Builder

NOTA

ALTERNATIVE A MAC OSX E XCODE

IBM ha rilasciato una propria guida allo sviluppo di applicazioni per iPhone, utilizzando come sistema operativo Windows o Linux. La guida si intitola *Write native iPhone applications using Eclipse CDT*. Si tratta di un PDF in lingua inglese che è possibile scaricare gratuitamente da questo indirizzo web: http://www.ibm.com/developerworks/edu/os-dw-os-eclipse-iphone-cdt.html. Per lo sviluppo vengono utilizzati Eclipse e Cygwin, tuttavia, le applicazioni sviluppate secondo questo schema non possono essere pubblicate sull'AppStore, ma soltanto su sistemi come *Cydia* e *Installer*.

al componente visivo: i file in questione avranno estensione *.h* e *.m*, e rappresenteranno quindi il Controller del nostro componente (in questo progetto non diamo attenzione al Model). Questi due file sono complementari, e rappresentano un'unica classe realizzata con un linguaggio object oriented (categoria alla quale ovviamente l'Objective-C appartiene); con l'estensione *.h* si indica l'interfaccia della classe, rappresenta quindi quell'insieme di metodi e variabili che si desidera rendere disponibili esternamente, alle altre classi e a Interface Builder.
Ecco come si presenta l'interfaccia del viewcontroller, il file *.h*, realizzato automaticamente quando si crea un nuovo progetto *View-Based Application*:

```
#import <UIKit/UIKit.h>
@interface ioProgrammoArt2ViewController :
                                    UIViewController {}
@end
```

Mentre quello che segue è il file *.m*, sempre creato automaticamente quando si crea il nuovo progetto, sarà:

```
#import "ioProgrammoArt2ViewController.h"
@implementation ioProgrammoArt2ViewController
/*
- (id)initWithNibName:(NSString *)nibNameOrNil
                    bundle:(NSBundle *)nibBundleOrNil {...}
- (void)loadView {}
- (void)viewDidLoad {... }
*/   /*
-(BOOL)shouldAutorotateToInterfaceOrientation:
            (UIInterfaceOrientation)interfaceOrientation {
  return (interfaceOrientation ==
                            UIInterfaceOrientationPortrait); }
*/
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning]; }
- (void)dealloc {
    [super dealloc]; }
@end
```

Quasi tutti i metodi sono commentati e, a seconda delle necessità, potrete decommentarli, o anche rimuoverli (non rimuovete gli ultimi due) vedremo nel prossimo articolo i loro possibili utilizzi.
È all'interno di quest'ultimo file che scriveremo il codice che gestirà qualunque tipo di operazione; ogni volta che inseriremo la signature di un determinato metodo all'interno del file *.h* è logicamente necessario creare la sua implementazione, quindi il suo codice effettivamente funzionante, all'interno del file *.m*. In questo caso è stata creata (automaticamente) una classe associata ad un viewcontroller, che utilizzeremo per gestire l'interazione con l'utente, ma è possibile anche crearne un'altra derivante da UIView e associarla alla view contenuta nel viewcontroller, e demandare a questa la gestione dell'interazione con l'utente (ricordiamo che qualunque componente visuale può avere associata una classe identificata da un file *.h* e *.m*); l'affermazione precedente è valida poiché entrambe le classi derivano da una comune, chiamata *UI Responder*, che consente di monitorare la pressione da parte dell'utente fornendo ad entrambe alcuni metodi: *touchesBegan, touchesCancelled, touches Ended, touchesMoved*. Il viewcontroller intercetta ogni input (il tocco con una o più dita), poiché in questo progetto la nostra UIView è utilizzata semplicemente come contenitore di altri componenti visuali (infatti non è stata creata alcuna classe che ne rappresenti il suo Controller del pattern MVC). Se vogliamo quindi consentire l'interazione umana con il ViewController, che contiene la nostra View, dovremo effettuare alcuni passi utilizzando IB. In primis **c**reeremo un file *.xib* (in questo caso è stato creato automaticamente), poi due file *.h* e *.m,* che estendono la classe che deve essere associata al componente grafico (anche in questo caso è stato creato automaticamente), nel nostro caso un *UIViewController*. Sarà quindi necessario intercettare la pressione dell'utente, oppure richiamare uno o più metodi se è stato premuto un pulsante o altro componente interattivo (keyword *IBAction* oppure adoperando *delegate*). Se desideriamo accedere facilmente nei nostri metodi a uno, o più elementi (pulsanti, label, webview etc) presenti nel ViewController dovremo, utilizzando la keyword *IBOutlet,* associare una variabile nel file *.h* a tale elemento grafico. Il passo successivo consisterà nel collegare all'interno di Interface Builder questa nuova classe con il componente visuale (facendo uso dell'Identity Inspector), quindi collegare (opzionalmente) quali metodi vengono richiamati quando l'utente interagisce con un determinato pulsante/componente interattivo. Sembra complicato ma dopo qualche incertezza diventerà un automatismo che eseguirete con estrema velocità.
L'operazione di associazione di un compoenente visuale a una classe è molto semplice: basta selezionare il componete grafico prescelto nel *Document Window* e successivamente selezionare la classe, precedentemente creata, all'interno del campo *Class*, contenuto nell'*Identity Inspector*. Nel prossimo articolo inseriremo un pulsante, una label e una webview, collegandoli tra loro utilizzando *IBAction* e *IBOutlet*, mostreremo l'alternativa fornita dall'utilizzo di *delegate*, gestendo tutto all'interno del file *.m* del ViewController. Forniremo inoltre una descrizione delle diverse tipologie di componenti grafici presenti.

Andrea Leganza

L'AUTORE

Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali e non su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certificatd Expert, EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multidediali con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it.

APPLE IPHONE PROGRAMMING

CONTINUIAMO IL CORSO INERENTE LA PROGRAMMAZIONE DEL DISPOSITIVO SMARTPHONE APPLE IPHONE, ILLUSTRANDO GLI STRUMENTI NECESSARI PER COMPLETARE L'APPLICAZIONE MINI BROWSER AVVIATA NEL NUMERO PRECEDENTE DELLA RIVISTA

CD WEB
corsoiphone2.zip
cdrom.ioprogramma.it

Nell'articolo del numero precedente abbiamo creato un progetto *View-Based Application* utilizzando XCode, fornendo, inoltre, una breve descrizione di Interface Builder; in questo nuovo appuntamento, posizioneremo i componenti grafici, un bottone, un campo di testo e una UIWebView, ovvero un componente visuale che si comporta come un browser web, al pari del software Safari installato nell'iPhone. Seguiremo le procedure necessarie per rendere accessibili tali elementi all'interno della classe (file *.h* e *.m*) del ViewController, scrivendo infine il codice utile per farli interagire e rispondere agli eventi generati dal tocco dell'utente. L'applicativo consentirà, al termine di questo articolo, di digitare un qualunque indirizzo web nel campo di testo, rendendolo così visualizzabile nella *UIWebView*.

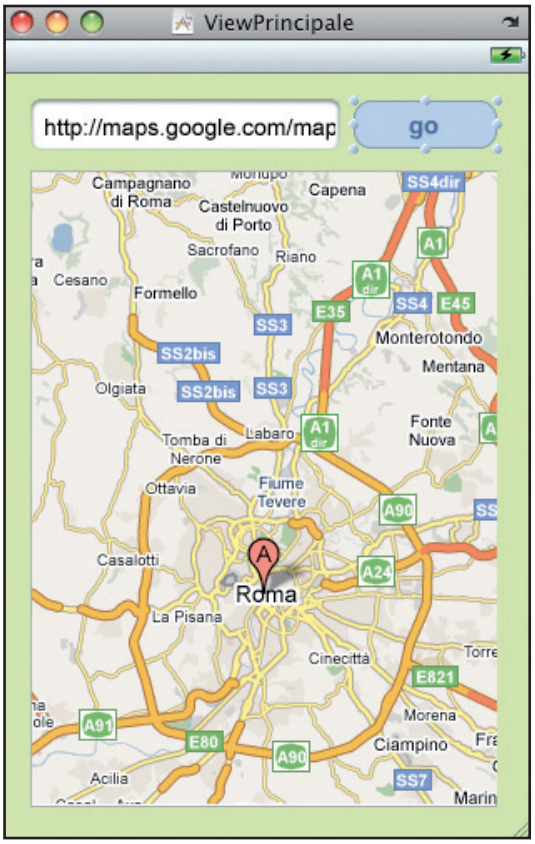

Fig. 1: L'interfaccia che realizzeremo in questo articolo

DISPONIAMO GLI ELEMENTI GRAFICI

Avviamo Interface Builder facendo doppio clic sul file *ioProgrammoArt2ViewController.xib*, ci si presenterà a video l'interfaccia di Fig.2
Trasciniamo i tre componenti all'interno della UIView e ridimensioniamoli utilizzando i punti

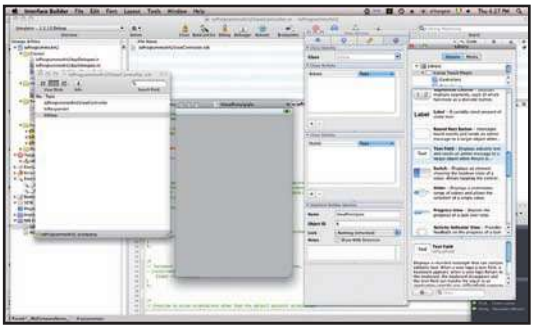

Fig. 2: Come si presente il file xib senza componenti aggiuntivi

presenti ai bordi: alcuni componenti non possono subire variazioni in alcune direzioni, come avviene per il campo di testo che può variare di lunghezza e non di altezza.
Per il bottone basterà fare doppio clic sull'area che gli compete per impostare la stringa che

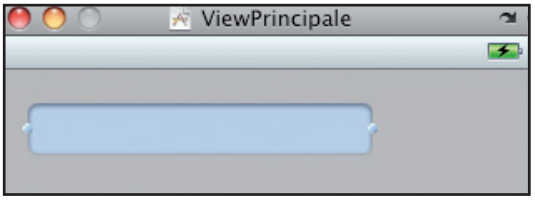

Fig. 3: I punti utilizzabili per il ridimensionamento di un campo di testo

REQUISITI

Conoscenze richieste
OOP

Software
Mac OS X 10.5 o superiore

Impegno

Tempo di realizzazione

verrà visualizzata come label, oppure aprire l'Attribute Inspector (*CTRL +1*) e impostare il campo *Title*; per impostare il colore verde della view principale basta selezionarla e cliccare sul tasto *Background*, che in questo caso ha assunto valori RGB *201,255,156*. La classe di appartenenza di componenti visuali appena inseriti, la potrete identificare analizzando la colonna *Type* del Document Inspector di IB: *UITextField, UIWebView* e *UIButton*; queste tre informazioni ci permetteranno di creare delle variabili ad-hoc

Fig. 4: La classe dei singoli componenti è visibile in corrispondenza della colonna type

nel prossimo paragrafo per utilizzarle nel codice Objective-C.
Da questa immagine si ponga anche attenzione al tipo di struttura che abbiamo creato.
La *UIView* principale, quella associata al nostro *UIViewController*, risulta la radice di un albero i cui tre figli sono proprio i tre componenti che abbiamo trascinato al suo interno.
Parliamo ora di *IBOutlet* e *IBAction*, keyword che ci agiscono come bridge tra Interface Builder e XCode.

COS'È IBOUTLET

Ora che abbiamo disegnato quella che sarà la nostra interfaccia grafica, dobbiamo realizzare il codice necessario per il funzionamento del nostro applicativo, quella responsabile del Controller, scrivendo istruzioni Objective-C all'interno della classe chiamata *ioProgrammo Art2ViewController.m*; prima di fare ciò è necessario spiegare come creare un collegamento tra i componenti visuali e oggetti in linguaggio Objectice-C, oggetti con i quali potremo interagire effettuando chiamate via codice: è in questa fase che viene creato un collegamento esplicito tra View e Controller. Il modo più semplice consiste nell'effettuare una procedura costituita da due semplici passi: creare tante variabili quanti sono i componenti grafici con i quali vogliamo comunicare, tre in questo caso, all'interno del file responsabile dell'interfaccia della classe, quello con estensione *.h*, anteponendo alla classe di appartenenza usata nella dichiarazione della variabile, il termine *IBOutlet*, acronimo di Interface Builder *Outlet*; *IBOutlet* è un qualificatore di tipo utilizzato solo per questo scopo, è quindi una semplice macro vuota corrispondente alla riga in linguaggio C *#define IBOutlet*; tale istruzione, quando letta da Interface Builder, lo informa che vogliamo realizzare una connessione tra variabile Objective-C e oggetto visuale:

```
//file ioProgrammoArt2ViewController.h
#import <UIKit/UIKit.h>

@interface ioProgrammoArt2ViewController :
                                    UIViewController
{

        IBOutlet UIWebView *webView;
        IBOutlet UITextField *addressField;
        IBOutlet UIButton *goButton;

}
@end
```

Le tre variabili hanno come classe di appartenenza la stessa dei componenti visuali, che potrete identificare sotto la colonna *Type* del Document Inspector di IB, come è stato mostrato nel paragrafo precedente. I tre IBOutlet sono caratterizzati da una tipizzazione forte *(strongly typed)* proprio a indicare che a queste variabili possono essere associati solo elementi grafici delle classi da noi indicate; una versione meno restrittiva (*weakly typed*) verrà mostrata successivamente, quando forniremo una versione più generica di *IBAction* (che nel prossimo articolo verrà spiegato approfonditamente). Le variabili create hanno come modificatore di accesso di default *@protected*, inaccessibili all'esterno, ma utilizzabile dalle classi che derivano dalla nostra (che in questo contesto è ininfluente, in un articolo successivo, descriveremo, tutti i modificatori disponibili). Questa prima parte della procedura ha un effetto immediato nella finestra delle proprietà del componente visuale chiamato *File Owner*, al quale è associata la classe *ioProgra mmoArt2ViewController*; accedendo quindi all'Identity Inspector (*CTRL+4*) ritroveremo le tre variabili, all'interno della sezione chiamata *Class Outlets*, esposte dalla nostra classe.
La seconda, e ultima parte della procedura, consiste nel premere il tasto destro del mouse dopo aver selezionato la riga corrispondente al File Owner, a cui è associata la classe *ioProgrammo Art2ViewController*: tale operazione presenterà una finestra semitrasparente nera nella quale troveremo di nuovo le tre variabili appena

NOTA

IL COMPONENTE UIWEBVIEW

Questo componente svolge la funzione di browser web, consentendo di visualizzare pagine internet nel nostro applicativo. Consente anche di consultare altri tipi di documenti, tra cui PDF, XML, immagini e altri: è possibile considerarlo una panacea in situazioni in cui si vuole svolgere il più velocemente un progetto, per limitazioni dell'API o per espressa volontà

NOTA

RIFERIMENTI WEB

Per la creazione dell'account e per scaricare l'SDK, visitate il seguente percorso web: http://developer.apple.com/iphone

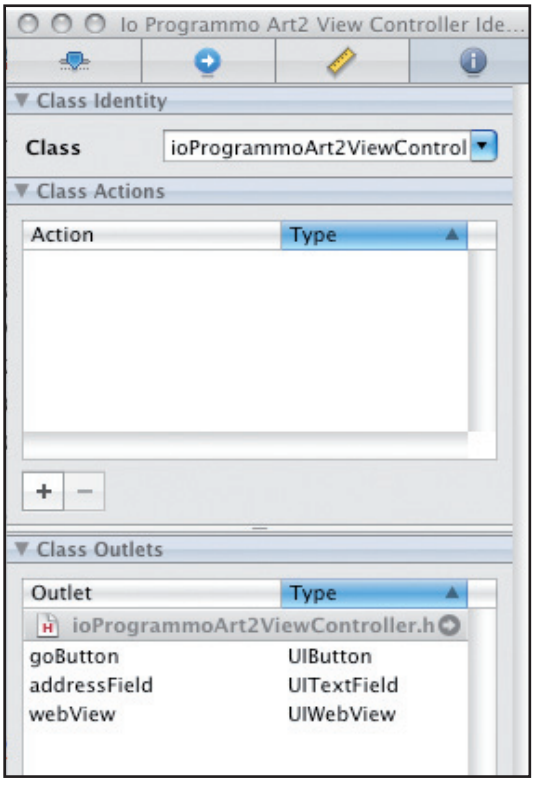

Fig. 5: Dopo aver inserito le tre variabili all'interno del file .h, queste saranno visualizzate nell'Inspector della classe in Interface Builder

aggiunte alla classe. Basterà trascinare il cerchietto, posizionato a destra, sul componente visuale che vogliamo associare.
Tale operazione deve essere ripetuta per le tre variabili, associando a ognuna il rispettivo com-

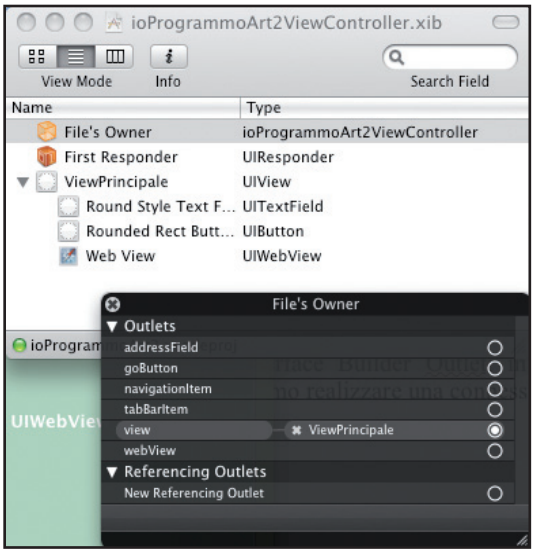

Fig. 6: La finestra semitrasparente

ponente visuale. Abbiamo così completato la procedura necessaria per accedere, nel file *ioProgrammoArt2ViewController.m*, a tali oggetti visuali e poterne così modificarne stato e aspetto; è l'iter più semplice da seguire quando si inizia a lavorare con i file *.xib*. È inoltre possibile invertire tale procedura lasciando che Interface Builder crei una classe partendo dall'interfaccia che avete creato, operazione che risulta di grande utilità quando si hanno molti oggetti visuali e non si desidera inserirli manualmente, oppure si è deciso di realizzare prima il file xib della classe associata: questa ulteriore procedura verrà spiegata, ma in un prossimo articolo.

LA COMUNICAZIONE TRA GLI OGGETTI

In ambito object oriented la comunicazione tra due classi avviene principalmente attraverso l'invocazione di metodi: in Objective-C tale operazione prende il nome di *invio di messaggi*.
Il design pattern, utilizzato per gestire gli eventi generati da un componente grafico, prende il nome di *target-action*; quando un utente interagisce con un componente lo induce a inviare un messaggio, chiamato *action*, a un altro oggetto, chiamato *target*, responsabile della sua gestione. È possibile procedere in modi diversi, a seconda della classe da cui derivano tali componenti: bottoni (*UIButton*) e campi di input di testo (*UITextField*) ad esempio, possono richiamare, dei metodi presenti in una istanza di una classe utilizzando il qualificatore di tipo *IBAction*, eseguendo una procedura molto simile a quella utilizzata nel paragrafo precedente (IBOutlet).
Per visualizzare l'elenco degli eventi che vengono lanciati da un componente visuale è sufficiente premere il pulsante destro sul relativo oggetto presente nel Document Inspector di Interface Builder; con altri componenti si adopera la tecnica del delegate; una terza soluzione, disponibile per i componenti che discendono da *UIControl*, come *UIButton*, *UITextField UIDate Picker* e altri, consiste nell'utilizzare i *selector*, adoperando istruzioni Objective-C; per ora tratteremo solamente di *IBAction*.

L'INTERFACE BUILDER ACTION - IBACTION

Con IBAction, acronimo di *Interface Builder Action*, identifichiamo quei metodi che Interface Builder mostrerà all'interno della propria interfaccia grafica quando è selezionata la classe in cui le abbiamo create, all'interno della sezione *Class Actions* del pannello *Identity Inspector di IB* (*CTRL+4*). Tali metodi rivestiranno il ruolo di gestori degli eventi lanciati dai

NOTA

IBOUTLET

IBOutlet viene utilizzato per notificare Interface Builder che tale variabile dovrà essere utilizzata all'interno dell'interfaccia grafica, svolge quindi la funzione di connessione tra Controller e View, instaurando un collegamento bidirezionale.

componenti visuali. IBAction viene utilizzato al posto del tipo di ritorno della funzione, è infatti una macro, sinonimo di *void* (*#define IBAction void*), e per tale motivo tali metodi non restituiscono alcun valore. Non è necessario creare una corrispondenza uno a uno tra un metodo IBAction e un evento richiamato da un singolo oggetto visuale, è possibile associare tale codice a un numero qualsiasi di eventi lanciati da altrettanti componenti visivi. Nel nostro progetto abbiamo necessità di gestire la pressione del pulsante, che provvederà in risposta a tale evento, a prelevare il valore del campo di input di testo e lo utilizzerà per aprire la pagina relativa all'interno del webView, per tale motivo la chiamiamo *gotoAddress*:

```
//file ioProgrammoArt2ViewController.h
#import <UIKit/UIKit.h>
@interface ioProgrammoArt2ViewController :
                                    UIViewController
{
   IBOutlet UIWebView *webView;
   IBOutlet UITextField *addressField;
   IBOutlet UIButton *goButton;
}

-(IBAction) gotoAddress;
@end
```

Abbiamo così esposto, semplicemente inserendola nel file di interfaccia e identificandola con *IBAction*, tale funzione in Interface Builder.
La funzione ci sarà presentata nell'area chiamata *Class Actions* di Interface Builder, nella sezione omonima dell'*Identity Inspector*:
Il passo successivo consiste nell'associazione della funzione alla pressione del pulsante che abbiamo creato, operazione identica a quella utilizzata con IBOutlet; basterà quindi premere il tasto destro sulla riga corrispondente a *File Owner* per notare la disponibilità all'interno della categoria *Received Actions*

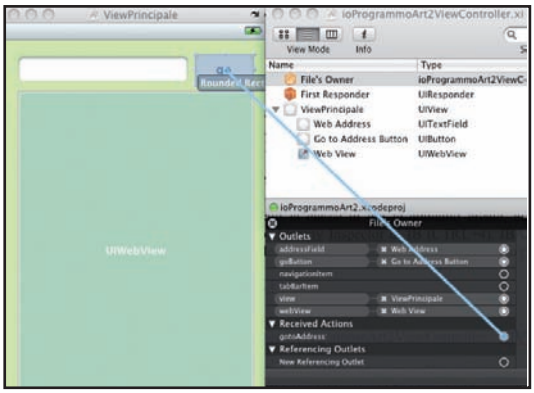

Fig. 7: La presenza dell'IBAction è ora mostrata anche in questa finestra

A questo punto la domanda potrebbe sorgere spontanea: perché viene inserito in questa categoria, e non sotto *Class Actions*, così come avviene nell'*Identity Inspector*? Questo perchè la pressione del tasto effettua una chiamata al nostro metodo e, dal punto di vista della classe, tale chiamata significa ricevere un'azione (un messaggio) da parte di un oggetto esterno, nel nostro caso un componente visivo; non fatevi confondere, quindi, sono sinonimi.
Trasciniamo il cerchietto sul pulsante, ci verrà presentato un elenco di eventi gestiti dal bottone, selezioniamo *Touch Up Inside*, (evento generato quando l'utente solleva il dito dallo schermo dopo aver premuto il nostro pulsante).
A questo punto non ci resta che scrivere il codice del metodo *gotoAddress*, ma prima sarà necessario spiegare come si invoca un qualunque metodo appartenente a un oggetto Objective-C.

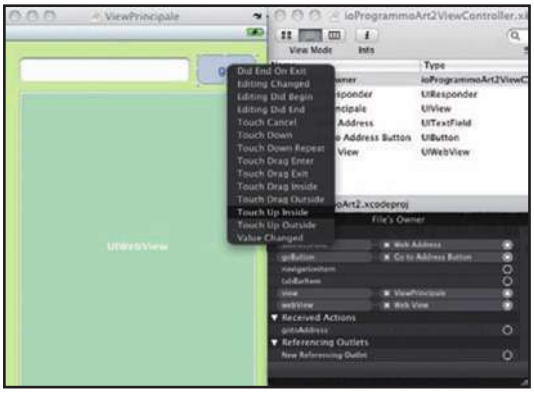

Fig. 8: L'effetto del trascinamento del cerchietto relativo all'IBAction sul pulsante

LA GESTIONE DEI METODI

Objective-C si differenzia principalmente da altri linguagg come Java, C#, Actionscript 3 poiché utilizza un approccio differente per il passaggio di parametri.
Per implementare un metodo è necessario inserire a priori la sua signature, costituita da tipologia, tipo del valore restituito, nome, due punti, elenco e tipo dei parametri, nel file di interfaccia, come abbiamo fatto per *-(IBAction) gotoAddress*, terminata dal punto e virgola, poi scrivere il metodo completo di codice nel file di implementazione, quello con estensione *.m*.
Un metodo che non accetta parametri verrà scritto all'interno del file di interfaccia (*.h*) nel seguente modo:

```
//file .h
- (void) nomeMetodo;
```

e verrà invocato tramite la sintassi:

VIEW-BASED APPLICATION

È il progetto più "comodo" per iniziare a programmare per iPhone/iPod Touch, infatti, fornisce una finestra contenuta all'interno di un controller, del quale viene fornita già una classe di implementazione che consente al suo interno di gestire la parte di Controller, del pattern Model View Controller, responsabile, quindi, della gestione degli eventi e dello stato del nostro applicativo.

```
//file .m
[oggetto nomeMetodo];
```

È quindi sempre necessario racchiudere tra parentesi quadre sia l'oggetto che il nome del metodo invocato. Nel caso restituisca un intero:

```
//file .h
- (int) nomeMetodo;
```

sarà possibile utilizzare tale risultato associandolo a una variabile nel seguente modo:

```
//file .m
int variabile = [oggetto nomeMetodo];
```

Il simbolo –, che precede ogni metodo, indica che questo è associato all'istanza, sostituendolo con un + lo si associa alla classe, verrà quindi invocato nel seguente modo:

```
//file .m
int variabile = [nomeClasse nomeMetodo];
```

In questi articoli, a meno di precisa motivazione, avremo sempre la necessità di creare metodi per consentire la chiamata a una istanza di oggetto, utilizzeremo perciò il -.
Esiste un modo per associare un modificatore di accesso (*public, protected, private*), a un metodo? Non esplicitamente, per rendere un metodo privato si possono usare le categorie, ma non è argomento di questo articolo: l'importante, per ora, è essere consapevoli che qualunque metodo è visibile all'esterno, da parte delle altre classi, a meno di utilizzare degli accorgimenti particolari, e che se non scriverete nell'implementazione della classe tutti i metodi dichiarati nell'interfaccia, verrete avvisati con il seguente warning:

```
warning: incomplete implementation of
                    class 'ioProgrammoArt2ViewController'

warning: method definition for '-
                                nomeMetodo:' not found
```

Nel caso un metodo accetti un parametro, si utilizza la seguente sintassi:

```
//file .h
-(void) nomeMetodo: (tipoSingoloParametro) par1;
```

Qui si nota la netta differenza con i linguaggi Object Oriented di "nuova" generazione; analizzandolo notiamo che i parametri seguono il simbolo dei due punti, poi si evince che non è presente un nome per il primo parametro, questo ha solo associati il tipo di appartenenza e il nome (alias) che assumerà all'interno del corpo del metodo. Il calcolo del fattoriale di un numero avrà questa signature:

```
//file .h
-(double) FattorialeDi: (int) par1;
```

Che sarà utilizzato in questo modo:

```
//file .m
int i = 5;
double risultato = [oggetto FattorialeDi:i];
```

Con più parametri, due in questo caso, si utilizza:

```
//file .h
-(void) nomeMetodo: (tipoPrimoParametro) par1
nomeSecondoParametro: (tipoSecondoParametro) par 2;
```

Se volessimo realizzare un metodo che effettua la somma tra due interi, si potrebbe utilizzare quindi la seguente sintassi:

```
//file .h
-(void) Somma: (int) par1 con: (int) par 2;
```

Da invocare nel seguente modo:

```
//file .m
int i = 100, j = 50;
int result = [oggetto Somma:i con:j];
```

È possibile effettuare chiamate in cascata come nel seguente esempio:

```
int result = [oggetto Somma:[oggetto Somma:i con;
                                        j] con: J];
```

Il cui risultato è la somma $(i+j)+j$; il numero delle chiamate è teoricamente illimitato, ma tale approccio ha il difetto di incrementare i rischi di errori per la necessità di porre attenzione alla posizione delle parentesi quadre, sia aperte che chiuse: XCode ci viene in aiuto, infatti, posizionando il cursore su una parentesi, verrà automaticamente evidenziata quella di chiusura corrispondente (o di apertura nel caso selezionassimo l'altra).

Fig. 9 Il sistema d'evidenziazione della parentesi quadra

NOTA

IBACTION

IBAction viene utilizzato per identificare un metodo di una classe, in modo da notificare Interface Builder che tale blocco di codice potrà essere invocato da un componente grafico; allo scaternarsi di uno o più eventi consente, inoltre, di esporre tale metodo all'interno di quelli forniti da un determinato oggetto visuale di IB. Ciò avviene quando si preme il destro sulla relativa riga nel document inspector.

IL CODICE DI GOTOADDRESS

Ora che abbiamo spiegato come invocare un metodo generico, dovremo semplicemente prelevare il testo del campo *addressField* e passarlo come indirizzo a *webView*. Creiamo il metodo all'interno del file *ioProgrammoArt2ViewController.m*, nel codice identificato dal blocco *@implementation ... @end:*

```
//file ioProgrammoArt2ViewController.m
#import "ioProgrammoArt2ViewController.h"
@implementation ioProgrammoArt2ViewController
-(IBAction) gotoAddress{
  //Corpo del metodo
}
@end
```

Per prelevare il testo dal campo di input e aprire la pagina web (se esistente), basterà utilizzare il seguente codice:

```
-(IBAction) gotoAddress{
//Preleviamo l'indirizzo
NSString *address = [addressField text];
//Creiamo un indirizzo utilizzando tale stringa
NSURL *url = [NSURL URLWithString:address];
//Creiamo una richiesta con tale indirizzo
NSURLRequest *request = [NSURLRequest
                                requestWithURL:url];
//Carichiamo la pagina utilizzando tale richiesta
[webView loadRequest:request];
}
```

Quando "cliccheremo" sul campo di testo si presenterà la tipica tastiera e, successivamente, alla pressione del bottone verrà caricata la pagina richiesta. È possibile raggruppare le chiamate effettuate in una sola riga:

```
[webView loadRequest:[NSURLRequest
 requestWithURL:[NSURL URLWithString:[addressField
                                          text]]]];
```

Come detto precedentemente, questa versione del codice risulta essere poco leggibile, a meno di non aver maturato una certa esperienza. Abbiamo completato la prima fase di questo semplice progetto. Ora, iniziamo a spiegare in dettaglio alcune caratteristiche dell'Objective-C e dell'interfacciamento con Builder, perché ci consentiranno di aggiungere ulteriori funzionalità nel prosieguo.

IBACTION E ID

Il precedente utilizzo di IBAction ha la limitazione di non fornire alcuna informazione sull'oggetto che lo ha invocato, nel nostro caso il bottone chiamato *goButton*, per tale motivo è disponibile una versione più completa, che nel nostro esempio sarà la seguente:

```
/file ioProgrammoArt2ViewController.h
- (IBAction)gotoAddress:(id)sender;
```

Abbiamo semplicemente aggiunto un parametro, di tipo *id* di nome *sender* che potremo utilizzare all'interno del blocco di codice del metodo per avere informazioni sull'oggetto che ha invocato tale comando. Questa funzionalità permette di condividere un IBAction tra un numero illimitato di oggetti, non solo di numero, ma anche di tipo, spetterà a noi analizzare la variabile *sender* per capirne la tipologia. Un tipico esempio di utilizzo è quello in cui abbiamo numerosi bottoni e desideriamo avere una sola zona di codice responsabile della loro gestione, utilizzando *sender* potremo identificare chi è il chiamante e, se necessario, modificarlo opportunamente.
Esiste un'ulteriore versione, la più dettagliata disponibile, nella quale abbiamo modo anche di identificare l'evento che l'ha invocata:

```
/file ioProgrammoArt2ViewController.h
- (IBAction)gotoAddress:(id)sender
                              Event:(UIEvent*)event;;
```

Tale variabile risulta utile se decidiamo di associare il metodo contemporaneamente a più eventi, utilizzando, ad esempio, *Touch Up Inside* e *Touch Down*, eventi "scatenati", in occasioni diverse, dal controllo afferente *UITextInput*.
Il "difetto" di tali metodi è dovuto al tipo del parametro utilizzato, *id*, il che rappresenta il tipo più generale possibile, che, se da un punto di vista consente grande libertà su quale componente utilizzare per invocare tale IBAction, dall'altro introduce alcune problematiche, principalmente riguardanti l'accesso ai campi e a i metodi dell'oggetto chiamante. Nel prossimo articolo a tale concetto dedicheremo particolare attenzione; l'utilizzo consapevole di *id* consente di accedere alla estrema versatilità fornita dal linguaggio Objective-C.

ALLA PROSSIMA...

Creata una versione funzionante dell'applicativo, nel prossimo articolo del corso, lo modificheremo per fornire ulteriori funzionalità. Faremo così conoscenza con altri argomenti correlati. Buona programmazione.

Andrea Leganza

L'AUTORE

Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali e non su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certificatd Expert, EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multidediali con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it

CREARE SOFTWARE PER L'APPLE IPHONE

IN QUESTA TERZA LEZIONE MODIFICHIAMO IL MINIBROWSER INTRODOTTO NEL PRECEDENTE ARTICOLO DELLA SERIE, INTRODUCENDO ALTRI CONCETTI FONDAMENTALI COME LA GESTIONE DEGLI ASPETTI GRAFICI E L'UTILIZZO DEL COMPONENTE UIWEBVIEW

CD WEB
iphonecorso4.zip
cdrom.ioprogrammo.it

Nel precedente articolo abbiamo completato un'interfaccia minimale e abbiamo realizzato la logica necessaria per il suo corretto funzionamento, ora ci dedicheremo all'introduzione di ulteriori caratteristiche del linguaggio Objective-C; aggiungeremo un'animazione per monitorare il caricamento della pagina web richiesta.

IL TIPO ID

Nella seconda versione del metodo *gotoAddress*, introdotto nell'articolo precedente, che qui riproponiamo per semplicità,

```
//file ioProgrammoArt2ViewController.h
(IBAction)gotoAddress:(id)sender;
```

abbiamo utilizzato un parametro di tipo *id*, che ci ha consentito di rendere il più generale possibile tale chiamata, in grado di essere invocata da qualunque componente visuale (ma non solo, e capirete il perché a breve): è necessario discuterne approfonditamente, poiché incontrerete *id* sia nell'API che nella documentazione; *id*, attenzione è privo dell'asterisco, simbolo utilizzato quando ci si riferisce alle istanze delle classi, esiste perché Objective-C è un linguaggio a *tipizzazione dinamica*, ciò significa che solo a runtime, in esecuzione, il vostro software avrà realmente modo di capire se una variabile appartiene ad un determinato tipo di dato, avrà quindi una struttura derivante alla sua catena di ereditarietà, partendo dalla root (in genere è *NSObject*), e scendendo progressivamente, risultando perciò corredato da precisi metodi e variabili; questo comportamento è completamente diverso da quello che presente in JAVA e C#, linguaggi a *tipizzazione statica*, dove tale controllo avviene in fase di compilazione, e durante il quale si riceve un errore nel caso si assegni a un'istanza di una classe un tipo diverso da quello della sua catena di ereditarietà (attribuendo ad esempio, il contenuto di una variabile *String* a una istanza di una classe creata estendendo un componente Swing) oppure si richiama un metodo inesistente (con parametri di tipo e/o numero diversi); in Objective-C se vengono rilevati dei conflitti o delle omissioni si riceve solamente un avviso, un warning in fase di compilazione, bisogna prendere consapevolezza che in tale contesto tutto è incerto riguardo una variabile tranne quando si è in esecuzione; solo gli errori sintattici, come parentesi e punti e virgola mancanti, interrompono il processo di compilazione. Associando, ad esempio, una variabile di tipo *UITextField* a una di tipo *UIWebView*, operazione concettualmente errata, ma perfettamente lecita in questo linguaggio:

```
UIWebView *oggettoBrowser;
UITextField *oggettoCampoDiTesto= oggettoBrowser;
```

riceveremo il seguente avviso:

```
warning: 'initialization from distinct Objective-C type'
```

Per i metodi il discorso è molto simile: il compilatore, se richiameremo un metodo inesistente, o sconosciuto al momento di compilazione come nel seguente caso:

```
//Definizione della variabile
UILabel *oggettoLabel;

//inizializzazione della variabile
...

//Chiamata del metodo
[oggettoLabel metodoInesistente];
```

segnalerà la mancanza di informazioni a riguardo con il seguente avviso

warning: 'UITextField' may not respond to 'metodoInesistente'

REQUISITI

Conoscenze richieste
OOP

Software
Mac OS X 10.5 o superiore

Impegno

Tempo di realizzazione

Notate il *may* utilizzato per denotare tale incertezza al momento della compilazione. Anche sbagliare la chiamata, invertendo "semplicemente" il tipo dei parametri, vi verrà segnalato come un warning, e non un errore, il compilatore presuppone che sappiate cosa stiate facendo e si aspetta, a runtime, di trovare questo metodo con tale diversa signature. Il vostro software verrà comunque compilato e avviato senza alcuna limitazione, il problema si presenterà a runtime e il risultato sarà uno, o più crash, con il conseguente avvio del debugger (spesso molto vago sulla causa scatenante del crash). Attenzione quindi ai warning del compilatore, perché sono il primo segnale di un errore semantico e quindi di un possibile crash! Da quanto detto, si ha un'estrema libertà (anche di sbagliare) nell'editor, ma si è di conseguenza poco "assistiti" dal compilatore per prevenire molti tra gli errori più comuni, che si erano risolti con i linguaggi di nuova generazione, per merito dei loro controlli preventivi.

Per merito di questa dinamicità intrinseca al linguaggio, è possibile utilizzare un "*placemark*", un indicatore, per segnalare che un determinato oggetto al momento della compilazione non appartiene a una classe predefinita, perché non è conosciuta a priori o non la si vuole specificare: in entrambi i casi si utilizza id come tipo della variabile. Nella documentazione relativa a *IBAction* viene utilizzato *id* per la signature, per indicarne la completa indipendenza dalla classe chiamante; *id* ha inoltre avuto il pregio di semplificare il processo di creazione dell'API, evitando di realizzare un metodo per ogni oggetto chiamante, poiché potrebbe appartenere a una qualsiasi delle decine di classi disponibili, *UIButton* e *UITextField* e via discorrendo; in casa Apple ci si è mantenuti il più generale possibile, lasciando quindi al programmatore piena libertà in che modo gestire tale chiamante. *Id* ha comunque delle limitazioni, a livello di usabilità: se si definisce una variabile di questo tipo non è possibile avere un elenco preciso, utilizzando il sistema di Code Completition, delle funzioni e delle variabili appartenenti a tale oggetto, poiché XCode non ha modo di sapere su che classe specifica si sta lavorando, riceveremo perciò un elenco esaustivo di tutti i metodi disponibili per tutte le classi presenti nell'API: migliaia di metodi, variabili e costanti, una lista di scarsa utilità!

Id può generare confusione nel codice se l'utilizzo dell'oggetto associato non viene opportunamente descritto, oltre a incrementare il numero di possibili bug, poiché si potrebbero chiamare metodi, e provare ad accedere a variabili, non presenti a runtime. Come è possibile limitare l'elenco dei metodi, e dei campi, suggeriti da XCode quando siamo certi del tipo di variabile che *id* assumerà a runtime? Nel nostro caso è del tipo *UIButton*, un bottone, passato come parametro sender nel metodo *IBAction*. La soluzione è molto semplice, basta creare una variabile del tipo desiderato e associarle il parametro *sender*; se pensiamo di utilizzarla un certo numero di volte, potremo utilizzare nel blocco di codice il seguente codice:

```
-(IBAction)gotoAddress:(id)sender  {

        UIButton *mioBottone = sender;

        [mioBottone metodoUno];
        [mioBottone metodoDue];
        ...
}
```

Oppure utilizzare il casting:

```
-(IBAction)gotoAddress:(id)sender  {
        [(UIButton *)sender metodoUno];
        [(UIButton *)sender metodoDue];
        ...
}
```

Quest'ultima tecnica dovrà essere utilizzata per ogni chiamata effettuata e/o per accedere ad un campo della classe, allo scopo di "forzare" l'editor a mostrare il corretto, e limitato, elenco di metodi e variabili; questa tecnica risulta però alquanto scomoda, degradando la leggibilità del codice, oltre ad incrementare il numero di caratteri da digitare. Se volessimo evitare di utilizzare *id* in questo contesto, potremmo semplicemente sostituirlo con il tipo corrispondente al componente grafico chiamante, *UIButton* nel nostro caso:

```
-(IBAction)gotoAddress:(UIButton *)sender  {
        [sender metodoUno];
        [sender metodoDue];
        ...
}
```

In questo modo l'operazione di casting verrà effettuata automaticamente. Applicando una qualunque di queste soluzioni, riceveremo suggerimenti coerenti con la classe che abbiamo specificato. L'unica limitazione dell'ultima soluzione è che impedisce la condivisione di tale *IBAction* con quei componenti che non appartengono alla classe *UIButton*, rendendo quindi impossibile qualunque associazione diversa da quella tramite Interface Builder. *Id* può essere utilizzato per qualunque variabile presente nella nostra classe, anche per gli *IBOutlet*; nell'articolo

NOTA

ID

id svolge la funzione di identificatore generico di classe; una variabile di tale tipo può appartenere a qualunque classe e la sua reale identità potrà essere evidenziata solo a tempo di esecuzione; ha il difetto di non fornire informazioni se uno o più metodi, o campi, a cui si accede esistono effettivamente a runtime, ciò incrementa il rischio di bug e crash del sistema, se non viene utilizzato con accortezza.

RIFERIMENTI WEB

Per la creazione dell'account e per scaricare l'SDK, visitate il seguente percorso web: http://developer.apple.com/iphone

precedente avevamo infatti detto che i tre IBOutlet utilizzati erano a tipizzazione forte, (*strongly typed*), cioè limitavano i tipi di componenti visuali che potevamo associare, evitando possibili errori di collegamento in Interface Builder. Con l'introduzione di *id*, possiamo ora rendere più dinamiche tali associazioni, basterà modificare il codice nel seguente modo:

```
IBOutlet id webView;
IBOutlet id addressField;
IBOutlet id goButton;
```

Questa versione, con *id,* chiamata anche a tipizzazione debole (*weakly typed*) è quindi la più generica tra quelle disponibili e consente di associare qualunque elemento visuale alle tre variabili: potremo quindi associare un campo di testo (visuale) al nostro webWiew (variabile) senza che il compilatore o Interface Builder segnali alcun problema; anche in questa situazione l'utilizzo indiscriminato di *id*, più che altro privo di sufficiente conoscenza, rischia di incrementare il numero di possibili chiamate a metodi o accesso a campi inesistenti, con la generazione di numerosi crash. Per soddisfare la curiosità di alcuni sulla reale identità di *id* bisogna consultare la documentazione e si verrà a conoscenza che è un semplice puntatore a una struttura che contiene la "classe", che verrà utilizzata a runtime:

```
typedef struct objc_object {
    Class isa;
} *id
```

Tale Class è un puntatore a una struttura:

```
typedef struct objc_class *Class;
```

definita nel seguente modo:

```
struct objc_class {
    struct objc_class *isa;
    struct objc_class *super_class;
    const char *name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
};
```

nella quale troviamo tutte le informazioni che rappresenteranno lo stato dell'istanza che id conterrà, tra cui metodi, variabili, protocolli, e altri dettagli; passando attraverso *id* siamo quindi giunti alla struttura che viene utilizzata per realizzare in Objective-C, utilizzando il linguaggio C, ogni classe che utilizziamo.
Può venire il dubbio, in alcuni contesti, se *NSObject*, poiché è la root delle classi che abbiamo fino ad ora utilizzato, è intercambiabile con *id*; la risposta a tale domanda è: dipende; *id* può ospitare qualunque istanza appartenente a una classe anche non discendente da *NSObject* (come *NSProxy* e *Protocol*); una qualunque chiamata a un metodo, o accesso a un campo, di *id* non viene verificata, e segnalata in caso di inesistenza o incongruenza, in fase di compilazione, mentre se attribuiamo a una variabile il tipo *NSObject,* riceveremo un warning se invochiamo uno, o più, metodi non esposti da tale classe di root; come è stato detto, il sistema di completamento del testo per *id* risulta praticamente inutilizzabile, presentando tutti i metodi disponibili nell'API, nel caso di NSObject, invece, riceveremo quei metodi presenti in tale classe, senza però ottenere quelli forniti dalla catena di ereditarietà: in entrambi i casi, per ottenere l'elenco più coerente di metodi è necessario effettuare il casting alla classe più specifica desiderata, come è stato mostrato negli esempi di *gotoAddress*.

NOTA

IDENTIFICARE LA REALE CLASSE DI ID

La necessità di utilizzo di *id* è relativamente ridotta, conviene adoperarlo in situazioni di indeterminatezza a runtime; per identificare la reale classe di una sua istanza, basta invocare su di essa (più correttamente "inviare un messaggio") uno dei seguenti metodi:

```
- (BOOL)isKindOfClass:
(Class)aClass
```

e

```
-(BOOL)isMemberOf
Class:(Class)aClass;
```

questi due consentono di ottenere un riscontro immediato, fornendoci le informazioni necessarie per effettuare un casting allo scopo di scegliere operazioni coerenti con la reale identità dell'istanza.

DELEGATE

Con *delegation*, o *message forwarding* (conosciuto anche con i nomi di *consultation* e *aggregation*), si intende quella procedura che ha come risultato l'indirizzamento degli eventi che una classe genera verso un'altra, presente al proprio interno come una variabile, e che viene configurata per gestirli opportunamente; tale nome è usato per descrivere un design pattern, descritto in ingegneria del software, che ha proprio questo comportamento; con delegate si intende una proprietà della classe, identificata con una variabile di tipo *id*, che viene utilizzata per realizzare la delegation. Non tutte le classi forniscono tale variabile, nel nostro progetto è disponibile per l'*UITextField* e *UIWebView*, ovviamente è la documentazione la fonte migliore per determinare se la supportano. Quando utilizzare il delegate? Come abbiamo detto precedentemente, in questo contesto le classi svolgono la funzione di Controller dei componenti visuali, a volte però non è necessario, o non si desidera, creare una classe che realizzi il Controller per ognuno di questi elementi, ma si vuole, comunque, gestire una certa famiglia di eventi inviati da tali istanze; un tipico esempio è quello della *UITableView*, una tabella visuale che mostra una serie di dati in sequenza; in genere si fa delegare tutta la sua gestione, insieme alla visualizzazione delle proprie informazioni e

all'interazione dell'utente, al *UIViewController* che la contiene (utilizzando il protocollo *UITable ViewDataSource* di cui parleremo nel prossimo paragrafo). È possibile impostare il delegate in due modi: il primo consiste nell'utilizzare Interface Builder, premere il destro del mouse sulla riga corrispondente al componente visuale del Document Inspector, noterete una riga con tale nome, e potrete associare tale proprietà a un oggetto semplicemente trascinando il relativo cerchietto, come è stato fatto per gli IBOutlet
L'altro consiste nell'utilizzare un comando in linguaggio Objective-C:

```
miaIstanza.delegate = oggettoDelegato;
```

Il risultato è identico, l'unica differenza è che nel secondo caso si ha modo di controllare quando inizializzare tale associazione, mentre nel primo avviene in un non precisato momento durante la trasformazione del file xib, che, come detto precedentemente, è un file XML, nel codice Objec tive-C necessario per realizzare l'interfaccia grafica da inserire nell'eseguibile. Abbiamo così impostato che l'oggetto *miaIstanza* delega la sua gestione a *oggettoDelegato*. Impostare il delegate di un'istanza di una classe non è però sufficiente affinché tale delegato possa intercettare ed elaborare gli eventi: è necessario che tale classe implementi i metodi che verranno richiesti, diventi cioè conforme ad un protocollo. Il nome di tale protocollo, corredato del relativo elenco di metodi, lo si ritrova nella documentazione associata al parametro delegate della classe sulla quale volete applicare la delegation. Per semplificare questa procedura è disponibile, nel caso della suddetta *UITableView,* una componente visuale chiamata *UITableViewController* che consiste proprio in un *UIViewController* e una *UITableView* al suo interno, già configurato in questo modo.

I PROTOCOLLI

Il concetto di protocollo suonerà familiare a molti utilizzatori di linguaggi object-oriented di nuova generazione, perché in Java, come in C#, vengono chiamati *Interfacce*, mentre *Classi Astratte* in C++. Diventare conforme (a volte si utilizza il termine *adattarsi*) a un protocollo, nella pratica significa garantire che una determinata classe utilizza, ha quindi implementato, tutti i metodi dichiarati nel protocollo rispettando i nomi, il tipo e il numero dei parametri utilizzati; il comportamento interno dei metodi non interessa al protocollo, è demandato al programmatore. Un esempio è quello dell'*UITable ViewData Source*, citato nel paragrafo precedente quando parlavamo della *UITableView* associata a un *UIViewController*, questo espone i seguenti metodi (è solo un estratto):

```
@required
- (NSInteger)tableView:(UITableView *)table
        numberOfRowsInSection:(NSInteger)section;
- (UITableViewCell *)tableView:(UITableView *)
    tableView cellForRowAtIndexPath:(NSIndexPath *)
                                        indexPath;

@optional
- (NSInteger)numberOfSectionsInTableView:
                                (UITableView *)tableView;
- (NSString *)tableView:(UITableView *)tableView
        titleForHeaderInSection:(NSInteger)section;
- (NSString *)tableView:(UITableView *)tableView
        titleForFooterInSection:(NSInteger)section;
...
```

come si può notare immediatamente, da una prima analisi di tale codice, non viene inserito alcun blocco di comandi Objective-C all'interno dei metodi, ma troviamo semplicemente il nome e i parametri. Ogni protocollo è da considerare al pari di un "patto" con il quale si garantisce che si rispetta un certo standard, almeno riguardo i nomi ed i parametri. I protocolli vengono suddivisi in due categorie: *formali* ed *informali*: con i primi si indicato quelli che, utilizzando solo la keyword *@required*, obbligano l'implementazione di tutti i propri metodi, mentre con i secondi si fornisce libera scelta su quale sottoinsieme di metodi realizzare, indicazione che avviene utilizzando la keyword *@optional*; dichiarando la conformità di una nostra classe a uno o più protocolli formali richiederà un controllo da parte del compilatore in fase di compilazione, al contrario di quelli informali, che vengono completamente ignorati. Le interfacce utilizzate in Java e/o C#, sono quindi comparabili ai protocolli formali. Ma perché in Objective-C è possibile avere tale libertà di scelta? Merito del fatto che, come è stato detto precedentemente, è un linguaggio a *tipizzazione dinamica*: anche il controllo se una classe rispetta un determinato protocollo, avviene a runtime, utilizzando un procedimento chiamato *reflection*, tramite il quale la classe viene interrogata (o più correttamente si interroga) se ha certe caratteristiche. Da quando detto potremmo implementare tutti i metodi necessari per il protocollo *UITableViewData Source* senza dichiararlo esplicitamente, tanto, se vengono rilevati in fase di esecuzione verranno comunque invocati. Analizzando i metodi presenti nella classe di root, *NSObject*, troviamo il seguente metodo:

```
- (BOOL)conformsToProtocol:(Protocol *)aProtocol
```

NOTA

DELEGATE

Con *delegate* si identifica quella istanza di una classe responsabile della gestione degli eventi lanciati da un altro oggetto; risulta utile quando si realizzano interfacce con numerosi componenti visuali e si preferisce centralizzare il codice all'interno di un numero più ristretto di classi.

NOTA

DEASOCCIARE IL DELEGATE

Quando si imposta manualmente il delegate con oggetti che lanciano eventi asincroni, come può accadere nel nostro *UIWebView*, si rischia di incorrere in chash casuali se non si deallocano correttamente le risorse associate se si rimuove il controllore; tale evento accade frequentemente quando si utilizza un controller *UINavigationController*, conviene quindi rimuovere questo legame nel metodo *dealloc* o appena possibile;

il cui scopo è proprio quello di fornire una risposta certa se si rispetta un determinato protocollo; tale metodo viene invocato automaticamente numerose volte durante l'esecuzione del vostro applicativo in maniera completamente trasparente. Dichiarare esplicitamente che una classe è conforme a uno, o più, protocolli permette sia di sapere quali metodi sono obbligatori (nel caso dei formali), perché anche una sola omissione viene segnalata dal compilatore, ma soprattutto è una pratica di buona programmazione (valida sia per i formali che per gli informali), rendendo palesi le proprie intenzioni anche ad altri sviluppatori che potrebbero analizzare in futuro il vostro codice. Per dichiarare se una classe è conforme a un protocollo, utilizziamo ad esempio quello mostrato precedentemente, *UITable ViewDataSource*, è sufficiente inserire il suo identificativo, racchiuso tra i due simboli minore e maggiore, all'interno del file di interfaccia della classe, successivamente alla classe da cui deriva la nostra:

```
@interface ioProgrammoArt2ViewController :
                    UIViewController <UITableViewDataSource>
```

Per aggiungere ulteriori protocolli basta semplicemente separarli con una virgola.
UITable ViewDataSource è un protocollo che dichiara sia metodi obbligatori (due) che opzional (numerosi), saremo quindi obbligati a implementare solo i primi due, *numberOfRowsInSection* e *cellForRowAtIndexPath*. La frase con cui è stato terminato il paragrafo precedente è quindi parzialmente vera, è necessario dichiarare esplicitamente di essere conformi a uno o più protocolli formali, ma è buona pratica farlo anche per quelli informali. È importante disaccoppiare il delegate quando rimuoviamo le risorse associate a un *UIViewController* che contiene la nostra classe, tale operazione si effettua semplicemente impostandolo al valore *nil*:

```
miaIstanza.delegate = nil;
```

Si tratta di un'operazione necessaria, poiché, se vengono inviati messaggi al delegate quando questo non è più disponibile, in genere dopo essere stato deallocato, si verificherà un crash a runtime; un errore comune è quello che si presenta quando si analizzano i dati dell'accelerometro e non si effettua tale procedura di rilascio. Concludiamo questo argomento suggerendo di porre una certa attenzione nella fase di scrittura della signature dei metodi, poiché una qualunque differenza con la versione attesa non permetterà a questi di essere invocati quando necessario: la procedura più sicura consiste nell'effettuare una operazione di copia, prelevando le signature dalla documentazione del protocollo, e incolla, all'interno di XCode.

IL CARICAMENTO DELLA PAGINA

Nel nostro progetto decidiamo di monitorare il caricamento della pagina web mostrando, quando questo processo non è ancora terminato, un'animazione; per semplicità utilizziamo un *UIActivityIndicatorView*, un componente visuale che mostra un'animazione infinita (ma comunque interrrompibile), il tipico effetto di una spirale rotante, familiare agli utenti Mac.
Abilitiamo, tramite la finestra *Attributes Inspector* (*CTRL+1*) di IB, la voce *Hide When Stopped*, in tal modo nasconderemo automaticamente tale componente quando interromperemo la sua l'animazione. Per accedere nella nostra classe *ioProgrammoArt2ViewController.m* a tale componente visuale dobbiamo aggiungere, effettuando anche il collegamento in IB, un nuovo *IBOutlet* all'interno di *ioProgrammoArt2ViewController.h:*

```
//file ioProgrammoArt2ViewController.h
#import <UIKit/UIKit.h>
@interface ioProgrammoArt2ViewController :
                                        UIViewController {
  IBOutlet UIWebView *webView;
  IBOutlet UITextField *addressField;
  IBOutlet UIButton *goButton;
  IBOutlet  UIActivityIndicatorView *loadMonitor; }
@end
```

Ricordate di associarlo in Interface Builder, altrimenti non si avrà modo di accedervi nei metodi che realizzeremo. Per mostrare l'utilità della delegation, invece di realizzare una classe da associare alla *UIWebView*, andremo a impostare come delegate di questo componente visuale il nostro *UIViewController*; come precisato precedentemente, è conveniente impostare il protocollo, anche se non espressamente richiesto; consultando la documentazione veniamo a conoscenza che il suo delegate è del tipo *id<UIWebViewDelegate>*, dovremo quindi essere conformi al protocollo *UIWebViewDelegate*:

```
@protocol UIWebViewDelegate <NSObject>
@optional
- (BOOL)webView:(UIWebView *)webView
        shouldStartLoadWithRequest:(NSURLRequest
    *)request navigationType: (UIWebViewNavigation
                                        Type)navigationType;
- (void)webViewDidStartLoad:(UIWebView *)webView;
- (void)webViewDidFinishLoad:(UIWebView *)webView;
```

NOTA

IBACTION

IBAction viene utilizzato per identificare un metodo di una classe, in modo da notificare Interface Builder che tale blocco di codice potrà essere invocato da un componente grafico; allo scaternarsi di uno o più eventi consente, inoltre, di esporre tale metodo all'interno di quelli forniti da un determinato oggetto visuale di IB. Ciò avviene quando si preme il destro sulla relativa riga nel document inspector.

NOTA

DEASOCCIARE IBOUTLET

Ogni volta che si utilizza un *IBOutlet*, dichiarandolo all'interno del file .h e associandolo ad un componente visivo tramite Interface Builder, è necessario rilasciare la memoria che gli viene riservata a runtime; tale operazione viene effettuata invocando il metodo release sul relativo *IBOutlet* all'interno del metodo *dealloc*.

Fig. 1: L'aggiunta del componente UIWebView alla nostra schermata

```
- (void)webView:(UIWebView *)webView
                didFailLoadWithError:(NSError *)error;
@end
```

I metodi disponibili sono tutti opzionali, provvediamo quindi a implementare solo *webView DidStartLoad*, che viene lanciato quando ha inizio il caricamento di una pagina, e *webViewDid FinishLoad* che notifica il suo completamento.
Prima dobbiamo impostare il delegate tramite Interface Builder, operazione che come precedentemente spiegato, avviene trascinando il relativo cerchietto verso il *File Owner*,
Ora dobbiamo passare a XCode e modificare prima il file *ioProgrammoArt2ViewController.h*

```
//file ioProgrammoArt2ViewController.h
#import <UIKit/UIKit.h>
@interface ioProgrammoArt2ViewController :
            UIViewController < UIWebViewDelegate>{
  IBOutlet UIWebView *webView;
  IBOutlet UITextField *addressField;
  IBOutlet UIButton *goButton;
  IBOutlet  UIActivityIndicatorView *loadMonitor;
}
@end
```

aggiungendo <*UIWebViewDelegate*>, e successivamente creando i due metodi richiesti, copiando la signature dalla documentazione, nel file *ioProgrammoArt2ViewController.m*

```
//file ioProgrammoArt2ViewController.m
#import "ioProgrammoArt2ViewController.h"
@implementation ioProgrammoArt2ViewController
-(IBAction) gotoAddress {
[webView loadRequest:[NSURLRequest requestWithURL
          :[NSURLURLWithString:[addressField text]]]];
}
- (void)webViewDidStartLoad:(UIWebView *)webView {
  // }
- (void)webViewDidFinishLoad:(UIWebView *)webView {
 // }
...
@end
```

Basterà infine avviare l'animazione in un metodo e interromperla nell'altro: ciò è reso possibile nella classe *UIActivityIndicatorView* invocando i due metodi, privi di parametri, *startAnimating* e *stopAnimating*.

```
- (void)webViewDidStartLoad:(UIWebView *)webView
{
   [loadMonitor startAnimating];
}
- (void)webViewDidFinishLoad:(UIWebView *)webView
{
   [loadMonitor stopAnimating];
}
```

CONCLUSIONI

Termina così questa serie di articoli in cui abbiamo esposto alcuni dei concetti che creano maggiori problemi quando si realizza per la prima volta un applicativo per iPhone. Nei prossimi mesi mostreremo il funzionamento di altri componenti visuali, e forniremo ulteriori nozioni sul linguaggio Objective-C. Buona programmazione.

Andrea Leganza

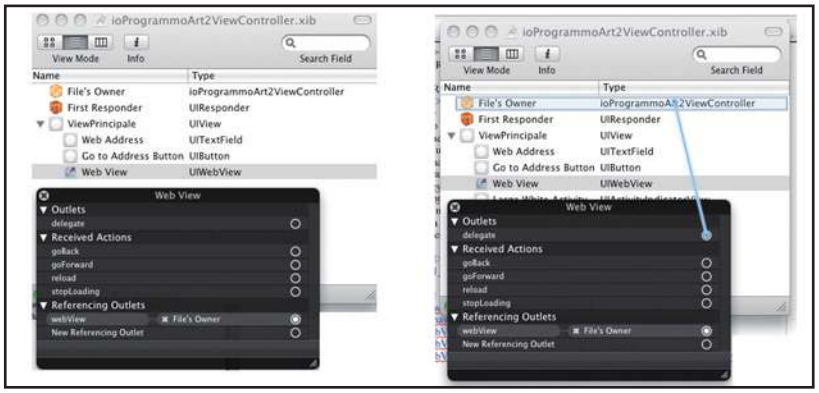

Fig. 2: Come si imposta il delegate

L'AUTORE

Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali e non su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certificatd Expert, EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multidediali con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it

REALIZZIAMO UNA AGENDA PER IPHONE

DA QUESTO NUMERO INIZIA UNA TRATTAZIONE CHE È UN PO' IL CUORE DI QUASI OGNI APPLICAZIONE IPHONE. STIAMO PARLANDO DELLE TABELLE. VEDREMO COME SI CREANO, QUALI TIPOLOGIE UTILIZZARE A SECONDA DEL CONTESTO, E COME GESTIRLE AL MEGLIO

CD WEB
ioProgrammoArt4.zip
cdrom.ioprogramma.it

Uno dei componenti più utili e utilizzati nello sviluppo di applicativi su 'iPhone, è quello identificato dalla classe *UITable View*, una tabella a scrolling verticale, che consente di mostrare un elenco ordinato, e opzionalmente modificabile da parte dell'utente, di informazioni generalmente coerenti. Tale componente è praticamente onnipresente in ogni applicativo, è possibile trovarlo in numerose personalizzazioni all'interno dell'elenco dei brani audio presenti nel vostro dispositivo, in quello dei film, in quello della posta elettronica, fino all'applicativo dei contatti. Lo utilizzeremo prima nella versione fornita, creando un progetto omonimo utilizzando il wizard e provvederemo poi a customizzarlo progressivamente per realizzare una nostra versione graficamente più accattivante. Il progetto che andremo a realizzare sarà una *todo list*, un normale elenco di azioni da fare, procedura che ci consentirà di analizzare in dettaglio la struttura di tale componente visuale, oltre che a prendere confidenza con le operazioni da effettuare per gestire l'interazione dell'utente e la relativa modifica, sia sotto l'aspetto grafico, che semantico delle informazioni che andremo a presentare visivamente: tutto ciò ci darà modo di introdurre nuovi tipi di dato e di conoscere tecniche e classi fornite da Objective-C.

Fig. 1: L'interfaccia che realizzeremo al termine della serie di articoli incentrati sulla tabella

TABELLE CON UITABLEVIEW

Come viene realizzata una tabella nella programmazione iPhone, all'interno dell'SDK?
La classe a cui fare riferimento è quella chiamata *UITableView*, le cui istanze ereditano a loro volta dalla classe da cui questa discende direttamente: *UIScrollView*, tale componente consente di effettuare lo scrolling di contenuti, ed è utilizzato normalmente quando la dimensione di

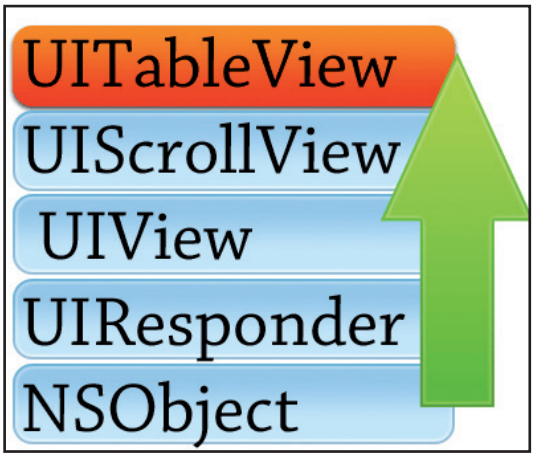

Fig. 2: La catena di ereditarietà della classe UITable

REQUISITI

Conoscenze richieste
OOP

Software
Mac OS X 10.5 o superiore

Impegno

Tempo di realizzazione

tali informazioni è superiore a quella concessa dallo schermo dell'iPhone/iPod (320x480 pixel massimo, ottenibili quando non si utilizzano componenti visuali di navigazione, come *UITabBarController* e *UINavigation Controller*); risulta molto utile per mostrare a video immagini di dimensioni relativamente grandi, oppure per fornire un semplice sistema di navigazione tra schede contenenti informazioni diverse; per mostrare la posizione in cui ci troviamo, vengono utilizzate due barre di colore normalmente nero, identiche a quelle utilizzate nel componente utilizzato nel precedente articolo, l'*UIWebView*. Rispetto a *UIScrollView, UITableView* aggiunge alcuni metodi e variabili specifiche per consentire l'interazione da parte dell'utente e per ottimizzare la visualizzazione e la persona-lizzazione, sia estetica che semantica, delle singole righe che vengono utilizzate per visualizzare le informazioni fornite; proprio le righe che abbiamo appena citato, in questo contesto prendono il nome di *celle* (sono per tale motivo istanze della classe *UITableViewCell*) e saranno oggetto principale di questa serie di articoli; tratteremo comunque in un numero successivo della rivista anche l'*UIScrollView*.

Poiché la classe *UITableView* discende, attraverso la classe padre, da *UIView*, ne eredita le caratteristiche, ed è necessario, per tale motivo, inserire questo componente sempre all'interno di un *Controller* o di una classe da esso derivata; oltre a effettuare tale pratica manualmente, utilizzando un qualsiasi *Controller*, inserendo la *UITableView* al suo interno, e gestendo la connessione tramite codice Objective-C (*IBOutlet*), è possibile velocizzare il tutto utilizzando un componente realizzato ad hoc, discendente di *UIViewController* e chiamato *UITableViewController.* Tale oggetto ha già al suo interno una variabile, una IBOutlet (tenete a mente le nozioni di *IBOutlet* e *IBAction* perché le ritroveremo sempre d'ora in poi, sarebbe opportuno andare a riconsultare gli articoli precedenti nel caso riteniate di non averli metabolizzati) che ospiterà la nostra tabella e per tale motivo risulta la strada più veloce da seguire quando si utilizza Interface Builder, evitandoci la creazione di una IBOutlet apposita. Inoltre, essendo alla sua radice una *UIView*, permette di essere personalizzata con componenti visivi, quali testi, immagini, bottoni etc. Se avete modo di analizzare con attenzione i diversi applicativi disponibili nativamente con l'iPhone /iPod Touch, troverete decine di varianti di questo componente, per non parlare delle versioni realizzate dai programmatori che pubblicano sull'Apple Store!

È probabilmente il componente più personalizzato e utilizzato fornito dall'SDK.

LA CREAZIONE DEL PROGETTO

Per semplificare questo tutorial, utilizzeremo una delle voci preesistenti presenti nel wizard dei nuovi progetti; selezioneremo per tale motivo un nuovo progetto della categoria *Navigation Based Application,* che ha il pregio di essere già configurato con una *UITableView*, inserita all'interno di un *UITableViewController.*

In questa circostanza non utilizzeremo la nuova funzionalità introdotta con l'SDK 3, chiamata *Core Data*, e, nel caso la trovaste selezionata (è attivabile utilizzando il checkbox che appare quando si seleziona questo tipo di progetto, ma solo se si ha aggiornato l'SDK alla versione 3) deselezionatela; per chi fosse curioso su cosa è *Core Data*, la risposta è semplice, è una nuova "funzionalità" che consente di gestire il *Model* senza dover scrivere comandi SQL per interfacciarsi con il database fornito dall'SDK: SQLITE: fornisce un livello di astrazione, che semplifica notevolmente l'utilizzo di questo database, esponendo metodi e metodologie il cui scopo è quello di ridurre al minimo l'impegno necessario per gestire la memorizzazione delle informazioni, che possono essere non temporanee (il cui tempo di vita è quindi superiore al singolo utilizzo dell'applicativo) e/o quando queste sono altamente strutturate (si pensi alla memorizzazione di informazioni di utenti se si realizzasse un sistema di interfacciamento con un social network): proprio per il loro particolare utilizzo dedicheremo in futuro opportuno spazio.

Quando avremo completato la fase di creazione del progetto troveremo alcuni file al suo interno, due *xib* (*MainWindow.xib* e *RootViewController .xib*) e alcune classi, tra le quali quella che ci interessa prende il nome di *RootViewController .h/.m*. Cerchiamo ora di capire cosa è stato creato dal wizard, ma in particolar modo come è stato realizzato.La procedura automatica ha creato un file *xib* che contiene (e conterrà) tutti i nostri componenti visuali, una *superview*, chiamata *MainWindow*, al cui interno viene inserito automaticamente il componente chiamato *RootView controller.xib*; questo file grafico è associato alla classe omonima che è un'istanza della classe *UITableViewController*: aprendo tale file all'interno di *Interface Builder* verrà visualizzata una tabella perfettamente funzionante, basterà infatti eseguire il progetto nel simulatore per effettuare lo scrolling utilizzando il dito. Anche se all'interno di Interface Builder, in questo caso, sono mostrati dei contenuti, delle città della California, non bisogna preoccuparsi di eliminare tali voci, andandole a cercare all'interno del codice del progetto, poiché queste sono semplicemente degli indicatori visivi di come potrebbe

NOTA

IPHONE 3GS

Il nuovo iPhone 3GS monta un processore di circa 600MHz, 256MB di RAM, OpenGL ES 2, contro il precedente di circa 412MHz, 128MB di RAM e OpenGl ES 1.1, l'iPod Touch 2G si pone nel mezzo con circa 533MHz e 128MB di RAM.

apparire la vostra tabella; non bisogna però tenere molto in considerazione tale anteprima, poiché non rifletterà più la reale visualizzazione della vostra tabella quando andrete a effettuare anche minimi cambiamenti al suo codice; se non effettuerete invece personalizzazioni utilizzando Objective-C, potrete effettuare modifiche utilizzando l'inspector (la finestra che contiene tutte le proprietà del componente visuale) per ogni modifica avrà un riscontro visivo immediato, anche se non sempre fedele. Avviare il progetto e testarlo, sia nel simulatore che sul dispositivo, sarà quindi l'unico modo per avere un riscontro reale dell'effettivo comportamento, e dell'aspetto reale, della tabella. Purtroppo, nonostante la relativa semplicità del progetto, perderemo di vista completamente Interface Builder, poiché non consente di effettuare modifiche estetiche (se non in maniera molto limitata) e strutturali alla tabella: utilizzeremo d'ora in poi solamente codice Objective-C.

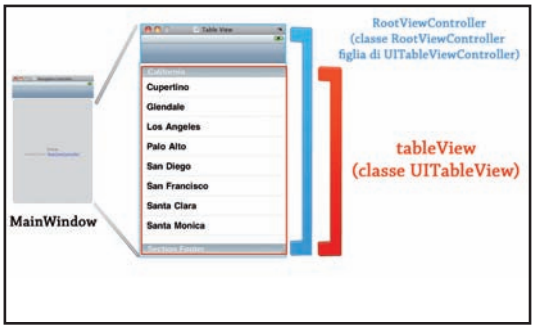

Fig. 3: Come sono strutturati i componenti visuali

NOTA

IBACTION

IBAction viene utilizzato per identificare un metodo di una classe, in modo da notificare Interface Builder che tale blocco di codice potrà essere invocato da un componente grafico; allo scaternarsi di uno o più eventi consente, inoltre, di esporre tale metodo all'interno di quelli forniti da un determinato oggetto visuale di IB. Ciò avviene quando si preme il destro sulla relativa riga nel document inspector.

LE RAPPRESENTAZIONI DI UITABLEVIEW

Esistono due tipi di rappresentazione grafica di una tabella standard: quella chiamata *plain,* che mostra un elenco semplice di righe, che d'ora in poi chiameremo con il nome utilizzato all'interno dell'SDK e della documentazione ufficiale, ovvero *celle*, e quella chiamata *groupe,* in cui insiemi di celle sono raggruppati in gruppi identificati da una stringa di testo. La prima modalità è la più semplice, e si presta a essere utilizzata quando non si desidera fornire una differenziazione tra diverse tipologie di informazioni mostrate a schermo, come per un elenco di dati appartenenti a una stessa tipologia, utenti di uno stesso social network a esempio, mentre la seconda diventa la soluzione logicamente, ma soprattutto graficamente, più opportuna quando mostriamo a schermo diverse tipologie di informazioni, elencando prodotti di un negozio ad esempio. La seconda modalità è utilizzata anche all'interno del menu *settings* dell'iPhone, dove nella stessa schermata troviamo valori e comandi utilizzati per configurare diversi applicativi e impostazioni.

La modalità *grouped* è inoltre la più adatta quando si raggiunge l'ultimo livello di dettaglio delle informazioni, quello in cui si mostrano tutti i dati riguardanti una determinata voce.

Decidere quale delle due tipologie scegliere, è quindi un'operazione relativamente semplice, tenendo a mente la loro diversa funzione.

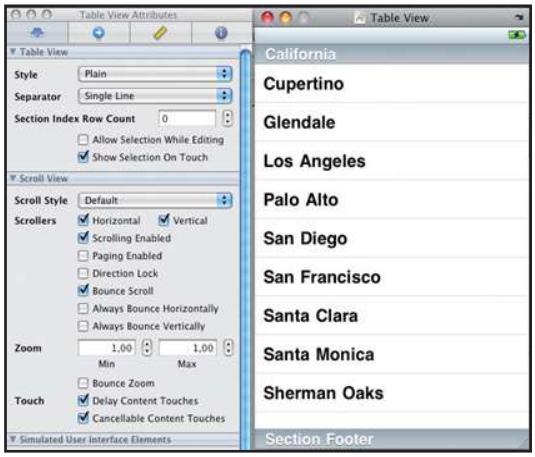

Fig. 4: La modalità di visualizzazione plain

LA STRUTTURA DELLA UITABLEVIEW IN MODALITÀ GROUPED

La modalità *grouped* merita un ulteriore approfondimento, poiché presenta una struttura particolare, la cui personalizzazione consente di variare il suo aspetto di default.

Nell'immagine di Fig.5 sono mostrati tre gruppi con realtive intestazioni, *Remote Host*: www.apple.com, *Access To Internet Hots* e *Access To Local Bonjour Hosts*, ognuno di questi ha una sola cella associata: in questo esempio possiamo chiaramente notare che viene creata una spaziatura sopra il gruppo chiamata *padding* (superiore),

Fig. 5: La modalità di visualizzazione grouped

poi troviamo uno spazio utilizzato per contenete l'*header*, l'intestazione del gruppo, a cui segue la cella vera e propria con i suoi contenuti, seguita da un *footer*, un'area in cui è possibile inserire testo ad esempio e infine da un ulteriore *padding* (inferiore). È possibile personalizzate tutte queste sezioni, ma solamente utilizzando codice Objective-C, si inizia a comprendere come Interface Builder perda di utilità anche nei contesti più semplici rendendo necessario familiarizzare con il linguaggio object oriented il più velocemente possibile.

CONSULTAZIONE MULTILIVELLO

Quando si fornisce una lista di informazioni presentate all'interno di una *UITableView,* generalmente si desidera rendere disponibile un sistema di consultazione multilivello, dove viene mostrato un primo livello, nel quale viene visualizzato un elenco complessivo, che l'utente potrà scorrere, e generalmente un secondo livello di accesso a tali dati, nel quale si mostrano i dettagli relativi alla cella selezionata nel livello precedente: è lo stesso principio applicato nella navigazione delle pagine web utilizzando i menu. In realtà il numero di tali livelli è virtualmente illimitato, ma superare di numero il valore tre/quattro crea problemi di orientamento all'utente che utilizza tale sistema di navigazione, molti utilizzatori non saranno in grado di ricordate percorsi superiori a tre interazioni e potrebbero trovare poco intuitivo tale approccio: il numero di utilizzi di un'informazione presente in un determinato livello si riduce in maniera inversamente proporzionale al numero di tale livello: potreste scoprire con il tempo che, tanta fatica fatta per realizzare una determinata UIView, non è apprezzato poiché troppo in profondità!

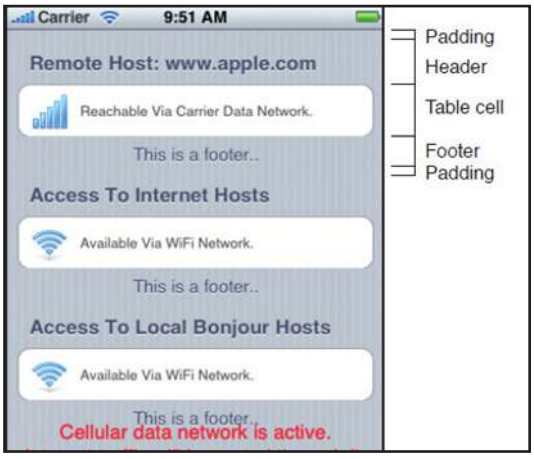

Fig. 6: Le diversa gestione degli spazi presente quando una tabella è del tipo grouped

Come detto in precedenza, quando si presenta l'insieme di informazioni relative all'ultimo livello, è consigliabile utilizzare una struttura di tipo *grouped* oppure, nel caso si desiderasse una struttura più complessa, una UIView (*UIScroll View* o simili) opportunamente strutturata.

GLI INDICATORI DI DETTAGLIO

Quando si realizza una struttura multilivello è necessario informare visivamente l'utente se esiste un successivo livello di navigazione cliccando su una determinata cella oppure no. Per tale motivo sono disponibili due icone per rappresentare la disponibilità di un maggiore dettaglio per l'informazione selezionata, e che vengono posizionate all'estrema destra delle singole celle. Una di queste, ottenibile impostando il tipo di cella come un *UITableViewCellAccessory Disclosure-Indicator*, e la cui immagine è una freccia verso destra, simile al simbolo di maggiore di colore grigio (>) chiamata *Dislosure Indicator*, informa che il prossimo livello generalmente conterrà un'altra tabella (con maggiore dettaglio), e un'altra, *UITableViewCellAccessory Detail-DisclosureButton*, chiamata *Detail Disclo sure Button*, identificata da un cerchio celeste con lo stesso simbolo di >, ma di colore bianco, che in questo caso indica che il prossimo livello dovrebbe essere quello finale; nessun controllo automatico vieta di agire in maniera diversa, realizzando diverse strutture con questi simboli, ma è altamente sconsigliato per evitare di non andare contro le regole di usabilità dettate dalla ditta di Cupertino che potrebbero impedire l'accettazione del vostro software per la vendita sull'Apple Store (è capitato a numerosi utenti): il consiglio è quindi quello di essere fedeli al comportamento standard quando si utilizzano simboli e i componenti forniti dall'SDK.
Ovviamente l'assenza di uno o dell'altro simbolo implica la mancanza di alcuna schermata successiva alla presente, è per tale motivo dimostrazione di scarsa attenzione non utilizzarli quando

Fig. 7: Un esempio di navigazione a tre livelli. Le informazioni sono sempre più dettagliate con il progredire del numero di livello

NOTA

PLAIN E GROUPED

La tabella di tipo *plain* si presta per elenchi generici, mentre quella *grouped* per rappresentare le schermate di dettaglio, oppure quando le informazioni mostrate sono raggruppate per una o più caratteristiche comuni.

necessario. È inoltre disponibile una terza icona, generalmente utilizzata quando si effettuano selezioni singole o multiple, quando si scelgono, ad esempio, diversi prodotti: è il simbolo di spunta (*UITableViewCellAccessoryCheckmark*) chiamato *Check mark*. Se questi sono i simboli più utilizzati, nessuno vieta, comunque, di personalizzarli a proprio piacimento, inserendo immagini create con il proprio programma di illustrazione preferita, renderete più accattivante e personale la vostra applicazione, evitando di farla sembrare troppo anonima e poco di impatto; conviene comunque utilizzare simboli di immediata comprensione per non confondere l'utente, evitando, ad esempio, di utilizzare frecce con verso illogico o immagini poco intuitive.

TIPOLOGIE DI CELLE

SDK 3.0 ha aggiunto un'utile proprietà per impostare il tipo di visualizzazione predefinita da utilizzare per le celle, nel caso non si volesse realizzare una propria versione.
Precedentemente era necessario posizionare i componenti visuali, utilizzando chiamate Objective-C, un'operazione che di ripeteva spesso in numerosi progetti, in questo modo si risparmia molto tempo, e si evita di incorrere in problemi di allineamenti e trasparenze errati.
Gli stili di visualizzazione sono i quattro seguenti: *UITableViewCell StyleDefault, UITableViewCell StyleSubtitle, UITa bleViewCellStyleValue1,UITable ViewCellStyleValue2*. Il primo tipo consente l'inserimento di un'immagine sulla sinistra della cella, un testo e un opzionale simbolo di dettaglio, il secondo aggiunge una didascalia sotto alla voce principale, mentre gli ultimi due sono solo elenchi testuali, non consentendo l'inserimento di alcuna immagine. Il testo principale è accessibile tramite la proprietà *textLabel,* mentre quello più piccolo utilizzando quello *detailTextLabel*.

IL SISTEMA DI NAVIGAZIONE MULTILIVELLO

Qualunque utilizzo di un applicativo iPhone è pensato o come lo sfogliare un libro, muovendosi parallelamente, oppure come navigare all'interno di un sito web, ottenendo informazioni sempre più dettagliate man mano che si scende in profondità

Fig. 8: I quattro tipi predefiniti di celle disponibili dalla versione 3.0 dell'SDK

DATA SOURCE E DELEGATE

Ogni tabella, istanza di *UITableView*, ha bisogno di essere associata a due altre istanze, di qualunque classe; infatti, controllando l'API, noterete che è utilizzato in entrambi i casi l'identificatore *id*, sinonimo di "qualunque oggetto"; una istanza sarà responsabile di fornirle i dati da mostrare a schermo, e viene identificata dal *Data Source* (Model del pattern MVC), e un'altra provvederà a gestire aspetto e comportamento (View e Controller del pattern MVC): non è necessario che queste due istanze siano diverse, infatti, in progetti non molto strutturati, si preferisce farli combaciare; tale comportamento viene inoltre seguito dal wizard stesso, che infatti ha provveduto automaticamente ad associare le due proprietà alla nostra istanza di *RootViewController*: basterà andare in Interface Builder, premendo il tasto destro del mouse sul *File Owner* per verificare tale affermazione. Da parte sua la *UITableView* creata riveste il ruolo per il RootViewController di View e di proprietà *tableView*. Si è realizzato quindi un ciclo di dipendenze. UITableView interrogherà i due delegati per effettuare tutte le operazioni, sia automatiche che scatenate dall'interazione dell'utente e su come presentare il proprio aspetto e quello delle celle. Data Source utilizza il protocollo *UITableViewDataSource,* mentre il delegato il *UITableViewDelegate*; nel primo caso si richiede al delegato, in questo caso il file R*ootViewController.h/.m*, di essere interme- diario tra la tabella e il model, fornendo su richiesta le informazioni necessarie e consentendo il loro inserimento, modifica e/o cancellazione. *UITableViewDelegate,* invece, è responsabile di fornire informazioni alla tabella su quale è l'aspetto delle celle e di come gestire l'interazione con l'utente, ed anche in questo caso coincide con *RootViewController.h/.m*; i metodi disponibili sono decine e la documentazione disponibile è la fonte più adatta.

IL PROCESSO DI CREAZIONE DELLE CELLE

Il procedimento necessario per la creazione di tutte le celle è relativamente semplice: la tabella interroga il proprio delegato, *UITableViewData Source*, richiedendo il numero di celle, sinonimo di righe, *rows*, utilizzando il metodo *numberOf RowsInSection*, per poi popolare le celle iterativamente invocando il metodo *cellForRow AtIndexPath*. Se non viene specificato, come avviene in questo esempio il numero di gruppi disponibili, identificati dal termine *sections*, e

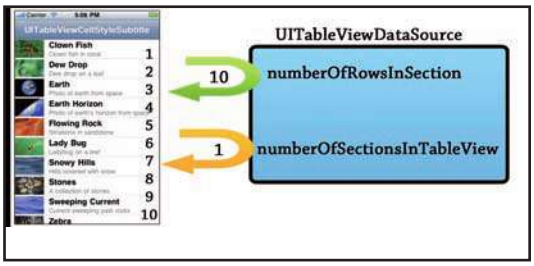

Fig. 9: La tabella richiede il numero di righe e il numero di sezioni che la stessa deve ospitare

ottenuto dalla tabella invocando sul delegate il metodo opzionale *sectionIndexTitlesForTableView*, tale valore è assunto come unitario e non verranno visualizzati header/footer e relative spaziature. Il metodo *cellForRowAtIndexPath* svolge quindi tutto il lavoro necessario per la generazione del componente visuale che verrà utilizzato per la tabella:

```
- (UITableViewCell*) tableView:(UITableView*)
tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
```

riceve per tale motivo come parametri la tabella, e le coordinate della cella, contenuti nella variabile *indexPath*, che nella sua versione più semplice fornisce solo la riga richiesta, nel caso di un semplice array monodimensionale, ma quando si utilizzano strutture dati più ramificate, questa è caratterizzata da numerosi indici (ad esempio 1.4.5), come avviene per i paragrafi di un libro, allo scopo di identificare univocamente la posizione delle informazioni richieste.

Tale metodo crea il componente visuale, un'istanza della classe *UITableViewCell* , figlia di *UIView*, configurandola attraverso l'inserimento di oggetti visuali, immagini e testi inclusi, richiedendo le informazioni al model (array o altra struttura dati): è qui che generalmente viene scritta la maggior parte del codice, anche se, di recente, con l'aggiunta dei tipi di celle predefiniti, fornita dall'SDK 3, si è notevolmente ridotta.

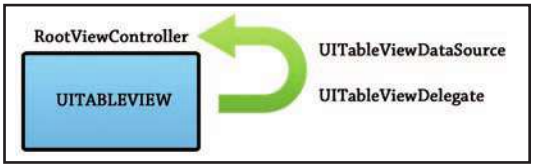

Fig. 10: La Tabella utilizza il file RootViewController come delegate per entrambi i protocolli

MIGLIORARE LE PERFORMANCE

Quando effettuate lo scrolling della tabella con un numero superiore a qualche decina, noterete inizialmente una fase di rallentamento iniziale, che successivamente non si presenterà.

Tale comportamento è dovuto alla necessità da parte del sistema di effettuare caching delle celle mostrate.

Nel caso andaste a visualizzare decine di celle in una sola schermata, potrete avere comunque fenomeni di rallentamenti, enfatizzati se avete personalizzato le vostre celle, aggiungendo immagini, testi, pulsanti, o altre UIView; ciò dipende da come gestirete i contenuti presenti all'interno della singola cella; preferite quindi contenuti senza trasparenza, ottenuti impostando la proprietà di opacità tramite Objective-C utilizzando *nomevariabile.opaque = YES* o tramite IB selezionando il checkbox *Opaque* (quando possibile).

Tale accorgimento consente di migliorare, spesso in maniera evidente, il rendimento a video durante la fase di scrolling: ridurre al minimo gli oggetti visuali trasparenti è quindi una buona operazione che non dovrebbe essere effettuata nella fase finale dello sviluppo; inserire, come è stato detto, un certo numero di oggetti, pulsanti, campi di testo e/o immagini, in una cella ne degrada le performance: in tal caso una soluzione consiste nel rasterizzare, convertendo quindi in immagine, l'intero contenuto della cella che si vuole mostrare a schermo, e inserire solamente questo all'interno della stessa; in questo modo si migliorerà notevolmente la prestazione complessiva del vostro software.

Un ulteriore accorgimento può essere quello di non creare una tabella per ogni livello di dettaglio, ma di modificare ogni volta sempre la stessa, in modo da ridurre al minimo il dispendio di risorse. Il nuovo iPhone 3GS, ha al suo interno un processore con circa 200 MegaHertz in più rispetto alla precedente versione, ciò significa un incremento effettivo di prestazioni comunque evidente, con relativo aumento di velocità di risposta nelle più comuni operazioni, testare quindi il vostro gioco o applicativo solo su tale dispositivo potrebbe essere un grande errore, poiché il numero di iPhone 3G/2G è al momento maggiore rispetto a quello di esemplari del nuovo modello: cercate di testare anche su uno dei due precedenti, in caso non risultasse notevolmente lento anche su questi dispositivi ne trarrete sicuramente vantaggio, sia di feedback positivi ottenuti dagli acquirenti, che di download, e in caso di vendita, di guadagni, perché ovviamente raggiungerete un numero più cospicuo di utenti interessati al vostro progetto..

Andrea Leganza

L'AUTORE

Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali, e non, su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certificatd Expert, EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multidediali con alcune società nazionali ed internazionali;
è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it

IPHONE: GESTIONE DELLA MEMORIA

PER CHI SI ACCINGE A SVILUPPARE APPLICAZIONI PER LO SMARTPHONE DI CASA APPLE, UNO DEGLI ARGOMENTI SICURAMENTE PIÙ OSTICI, È LA GESTIONE DELLA MEMORIA. IN QUESTO ARTICOLO AFFRONTEREMO PROPRIO QUESTA IMPORTANTE TEMATICA

CD WEB
ioProgrammoArt4.zip
cdrom.ioprogrammo.it

Nell'articolo precedente abbiamo descritto la struttura, intesa come insieme di componenti visuali, che viene creata automaticamente quando realizziamo, attraverso l'utilizzo nel wizard, un progetto del tipo *Navigation-Based Application*; è stato inoltre introdotto il componente chiamato *UITableView* e le relazioni che si instaurano tra la popolazione delle celle (sinonimo di righe della tabella) e il metodo *cellForRowAtIndexPath*, che, come spiegato, svolge tutto il lavoro necessario per la generazione di tali componenti visuali all'interno del delegate. Analizzeremo tale metodo in maniera dettagliata, in modo da introdurre alcune caratteristiche del linguaggio Objective-C che, potremo affermare senza alcun dubbio, essere le più importanti, e anche più impegnative da comprendere: per tale motivo è vivamente consigliato testare personalmente i vari aspetti trattati in queste pagine per metabolizzarli correttamente.

NIL, NIL, NULL E NSNULL

Ogni istanza di un oggetto, quando viene definita (il suo stato interno non è quindi ancora stato inizializzato) assume automaticamente valore *nil* (ad esclusione di *isa* che assume la struttura della classe di cui la variabile è istanza); in ambito Objective-C *nil* indica che una variabile (che dovrebbe puntare a un'istanza di un oggetto) non punta ad alcuna locazione di memoria. NULL, familiare a chi programma in C, è anche disponibile nel linguaggio Objective-C, ovviamente questo è dovuto al fatto che tale linguaggio è una versione opportunamente modifica del C. *nil* viene quindi utilizzato per i puntatori ad oggetti, mentre NULL per qualunque puntatore (esistono ad esempio quelli a funzioni); entrambi, comunque, se andiamo ad analizzare la loro effettiva semantica, assumono valore pari a (*void***)0*.

```
#ifndef NULL
#define NULL    __DARWIN_NULL
#endif /* ! NULL */
#ifndef nil
    #define nil NULL
#endif
```

Se utilizzerete sempre Objective-C incontrerete molto raramente *NULL*, a meno di accedere ad alcune funzionalità particolari, che generalmente scendono a livelli più bassi, in linguaggio C puro, quindi, è comunque necessario essere a conoscenza della differenza tra i due e dei diversi contesti in cui li potete incontrare. *Nil*, con la prima lettera maiuscola, è utilizzato per i puntatori a classi, e per tale motivo lo troverete ancora più raramente, poiché generalmente viene utilizzato lo stesso *nil* (sono anche questi sinonimi dal punto di vista sintattico).
NSNULL è presente invece in quelle classi che fanno parte della Foundation collection (*NSArray, NSSet, NSDictionary* e altre) poiché queste non possono contenere il valore *nil* . Se vogliamo verificare che un oggetto è appena stato creato, ma non ancora inizializzato, possiamo effettuare il seguente controllo:

```
if (myObject==nil)
{
    //inizializzazione
    ...
}
```

Sarebbe ideale effettuare tale controllo prima di ogni utilizzo di una variabile, ma risulterebbe dispendioso sia in termini di memoria/risorse sia per numero di righe di codice, conviene quindi utilizzarlo in quelle situazioni in cui si è dubbiosi sullo stato di una variabile, ad esempio quando questa viene creata in un metodo e utilizzata in un altro, e spiegheremo a breve perché tale tecnica può risultare di fondamentale utilità.
Inoltre, invocare qualunque metodo su tale variabile restituisce sempre 0 (in realtà quasi sempre ,ma non ci addentriamo troppo) e viene quindi

REQUISITI

Conoscenze richieste
OOP

Software
MacOS X 10.5.4 o superiore, XCode

Impegno

Tempo di realizzazione

interpretato come *false* nei controlli condizionali, *if*, *else*, *while* etc.
A differenza del linguaggio C, in cui invocare su una variabile *NULL* generalmente causa un crash dell'applicativo, in questo contesto si ottiene invece sempre un risultato da tali chiamate, anche se poi inutilizzabili nella pratica, oltre che sintomo di scarsa comprensione del funzionamento del linguaggio, e causa di dispendio di risorse, anche se modestissimo, ma che in un dispositivo con risorse relativamente limitate può fare la differenza, dovuto allo scatenarsi di tutto il sistema di invocazione del metodo relativo (anche se inesistente); è quindi perfettamente lecito effettuare la seguente chiamata:

```
NSString miaStringa;
miaStringa = nil;
[miaStringa isEqualTo:altraStringa];
```

Spesso, proprio utilizzare una variabile che assume il valore *nil* come parametro di un metodo, scatena una serie di problematiche che generalmente terminano con un crash del sistema: sono la causa più frequente di problemi quando si programma con tale linguaggio e per evidenziare tali situazioni è necessario generalmente utilizzare il debugger fornito, che, come scopriremo in un prossimo articolo, è "semplicemente" il famoso *gdb*, proveniente direttamente dall'ambiente GNU/Linux (ma non solo).
È possibile assegnare in qualunque momento *nil* quale valore di una variabile, ma in tal caso c'è il rischio di non rilasciare le opportune risorse associate a tale istanza (situazione identificata con il termine *leak*),

```
//effettuare prima la deallocazione delle risorse
                        associate alla variabile myObject.
myObject==nil
//qualunque chiamata successiva non avrà alcun
                                        risultato
//Non effettuare tale operazione se la variabile
                non è stata deallocata opportunamente!!!
    ...
}
```

le motivazioni di tale affermazione verranno spiegate successivamente, quando si introdurranno i concetti relativi alla gestione della memoria.
Per concludere questa sezione è opportuno ricordare che esistono altri valori che assumono valore 0, ma utilizzati in contesti prettamente grafici: *NSZeroPoint* (un punto di coordinate 0,0). *NSZeroSize* (una dimensione *NSSize* di larghezza e lunghezza 0) e *NSZeroRect* (un rettangolo *NSRect* posizionato nell'origine di di larghezza e lunghezza 0);

IL METODO CELLFORROWATINDEXPATH

Dopo aver descritto cos'è *nil* troverete sicuramente più semplice la lettura del codice del metodo *cellForRowAtIndexPath*, che qui per chiarezza presentiamo nella versione prodotta direttamente dal wizard:

```
- (UITableViewCell *)tableView:(UITableView
                    *)tableView cellForRowAtIndexPath:
                            (NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
     dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    }
            // Configurazione della cella
    return cell;
}
```

Generalmente, una tabella ottiene i dati da una struttura dati, un array ad esempio, in ambiente Objective-C identificato dalla classe *NSArray*, e dovrà mostrare, a meno di particolari condizioni, come un semplice filtraggio effettuato in una ricerca, tutti gli oggetti in essa contenuti e utilizzare i valori di ogni singola istanza per popolare le celle della tabella: se ogni oggetto conterrà al suo interno una stringa, potremo utilizzare tale variabile per mostrare nella cella un testo visibile all'utente che scrollerà la tabella.
Quando la tabella, come è stato spiegato approfonditamente nell'articolo precedente, richiede al proprio delegate la struttura delle singole celle, operazione che avviene invocando il metodo suddetto, effettua sempre alcune operazioni, obbligatorie, e presenti nel metodo *cellForRowAtIndexPath* prodotto dal wizard: viene creato un certo numero di celle (graficamente parlando), che non corrisponde al reale numero di oggetti presenti nella struttura dati: quando verrà effettuato lo scrolling della tabella tali celle verranno riutilizzate e ne verranno cambiati i valori interni, nel nostro caso, quindi, (ricordiamo che stiamo progettando un'applicazione agenda), l'azione da effettuare per una certa giornata: questo procedimento è stato ideato per ottenere un notevole risparmio di risorse, si pensi che cosa succederebbe se si creassero tutte le istanze delle celle anche quando l'utente non le visualizza: quasi ogni applicativo verrebbe chiusa per saturazione della memoria disponibile nel dispositivo. Con questa tecnica viene risolto questo possibile collo di

bottiglia in maniera molto intelligente; quella descritta non è comunque una tecnica presente solo in tale linguaggio: prende il nome di *UI Virtualization*, ed è possibile ritrovarla, ad esempio, anche in Adobe Flex 4 per il nuovo componente *DataGroup* (*Spark*), e in *Windows Presentation Foundation* per il componente *DataGrid* fornito nel *WPF Toolkit*. La variabile statica *cellIdentifier* viene creata solamente durante la prima invocazione del metodo e permette di identificare la tipologia di cella che stiamo creando, utilizzando una stringa, che in questo caso corrisponde al valore "*Cell*"; nessuno vieta di creare diverse celle con diverse caratteristiche e decidere, a seconda delle necessità, quale utilizzare per un determinato oggetto prelevato dalla nostra struttura dati.
Se volessimo quindi avere due celle diverse per qualche caratteristica, basterebbe effettuare la seguente operazione:

```
static NSString * CellIdentifierWhiteBackground =
                                              @"Cell";
static NSString *CellIdentifierBlackBackground =
                                  @"CellBlackBackground";
if (decisioneCellaNormale)
UITableViewCell *cell =
[tableView dequeueReusableCellWithIdentifier:CellI
                                  dentifierWhiteBackground];
{
//codice di formattazione grafica e
                                  popolazione della cella
}
else //decisioneCellaBlackBackground
UITableViewCell *cell = [tableView dequeueReusable
     CellWithIdentifier:CellIdentifierBlackBackground];
{
//codice di formattazione grafica e
                                  popolazione della cella
}
...
return cell;
```

Potremo inoltre conoscere il tipo di cella richiamando la proprietà (di cui parleremo successivamente in maniera approfondita, per ora identificatela come una variabile pubblica, accessibile quindi dall'esterno della classe) *currentCell.reuseIdentifier* che restituirà la stringa utilizzata (nel nostro esempio corrispondente a *CellWhiteBackground* oppure *CellBackBackground*). Proseguendo notiamo che prima viene verificato che non esiste già una cella con l'identificatore che abbiamo deciso ("*Cell*") , in caso non sia stata già creata viene invocato il costruttore della cella, utilizzando lo stile *UITableViewCellStyleDefault* (spiegato nell'articolo precedente) e tale istanza viene inserita nella *Autorelease Pool*.

ULTERIORI APPROFONDIMENTI

Per approfondire i concetti mostrati in questo articolo, consultare la guida *chiamata Memory Management Programming Guide for Cocoa* disponibile su sito developer.apple.com o all'interno della documentazione fornita con Xcode

LE REGOLE PER LA GESTIONE DEGLI OGGETTI

Queste regole sono i concetti fondamentali che devono essere sempre tenuti in considerazione durante lo sviluppo di qualunque applicativo, vengono introdotti solo ora perché negli articoli precedenti non erano utilizzati quegli aspetti del linguaggio che potevano indurre confusione e generare crash, inoltre si è alleggerito il contenuto del primo tutorial :

- Un oggetto può essere utilizzato da uno o più proprietari (altri oggetti);
- quando un oggetto non ha proprietari, viene distrutto automaticamente senza alcun preavviso e in un momento imprecisato durante l'esecuzione del vostro applicativo dall'Autorelease Pool, generalmente dopo l'uscita dal metodo in cui è stato creato;
- per impedire la sua distruzione automatica si deve notificare l'Autorelease Pool che si è divenuti proprietari di tale oggetto (utilizzando il metodo *retain*);
- se non si è più interessati a tale oggetto, si deve smettere di esserne proprietari (utilizzando i metodi *release/autorelease*) per consentirne una possibile distruzione da parte dell'Autorelease Pool;
- il numero delle dichiarazioni di appartenenza (*retain*) deve essere bilanciato con quello dei rilasci (*release*) in modo da consentire il rilascio automatico;
- il numero di dichiarazioni di utilizzo viene chiamato *retainCount* e assume superiori o uguali a 0, in caso assuma quest'ultimo verrà deallocato dall'Autorelease Pool;
- quando si raggiunge una certa esperienza è opportuno utilizzare il più possibile *alloc/retain/release* invece di utilizzare *autorelease*, per motivi prestazionali, soprattutto all'interno di cicli con un numero consistente di iterazioni;
- il sistema invia un avviso di risorse insufficienti (principalmente RAM) quando questa decresce raggiungendo un valore prossimo ad 1.5MB.

Se la prima regola è un concetto condiviso da tutti i linguaggi Object-Oriented (e non solo), ed è quella che consente di condividere una variabile tra più istanze di una o più classi, analizzando le altre si può intuire che esistono due metodologie per gestire la memoria occupata da una variabile, una automatica (*Autorelease Pool*) e una manuale (*alloc/retain/release*) dove è responsabilità del programmatore gestire il tutto; non sempre la modalità automatica risponde alle esigenze di sviluppo e per tale motivo entra in gioco la seconda.

AUTORELEASE POOL

L'Autorelease Pool nasce come necessità, in ambiente Objective-C di fornire un sistema automatico di rilascio delle risorse non utilizzate, che molti potrebbero identificare con il sistema di Garbage Collection (GC) presente in numerose VM, Java e .NET in primis; poiché nel dispositivo Apple non viene eseguito un vero motore di GC, principalmente per motivi prestazionali e di occupazione di risorse, quando avviamo i nostri applicativi (ricordo che alla fine sono scritti in linguaggio C) questa è la soluzione automatizzata che ci viene fornita.

Come funziona l'Autorelease Pool? Tale componente è identificabile come un analizzatore di istanze di oggetti sulle quali è stato invocato il metodo *autorelease*, sono stati quindi identificati come oggetti il cui ciclo di vita viene deciso dall'Autorelese Pool stesso; questo provvede a liberare la memoria, distruggere quelle istanze di oggetti non più necessari, in maniera automatica, analizzando una particolare proprietà chiamata *retainCount* (un variabile numerica presente in ogni istanza il cui funzionamento verrà spiegato a breve): quando troverà questa pari al valore 0. L'autorelease pool invoca sulla variabile un metodo, che viene identificato come il suo distruttore, chiamato *dealloc*, nel quale si effettuano generalmente operazioni di rimozione di legami con altre variabili (utilizzando il comando *release/dealloc* spiegati in seguito).

Durante il ciclo di vita di una variabile, il *retainCount* viene quindi incrementato e decrementato, fino generalmente a terminare in un determinato istante *t*, in cui questo assumerà valore 0: non è detto che si raggiunga tale valore durante l'utilizzo del software, è ammissibile che una variabile abbia *retainCount* superiore a 0 per tutta la sua esistenza, e termini la sua vita alla chiusura dell'applicativo.

L'operazione di rilascio automatico avviene in maniera non prevedibile e non è possibile quindi sapere quando viene effettuata in maniera automatica, a meno di notificare tale evento utilizzando le stampe a schermo all'interno del metodo *dealloc*, generato automaticamente in ogni classe da parte del wizard; questa operazione è realizzabile semplicemente aggiungendo una stampa a schermo prima dell'invocazione del metodo distruttore sulla classe padre:

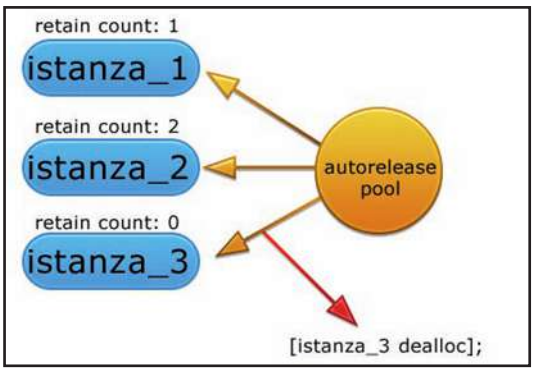

Fig. 1: L'Autorelease pool distrugge le istanze delle classi con retainCount pari a 0 automaticamente, invocando su di esse il metodo dealloc()

```
-(void) dealloc {
NSLog(@"Istanza deallocata");
[super dealloc];
}
```

In questo modo, osservando la finestra di log quando si avvia un applicativo in modalità di debug, si avrà modo di sapere con una certa precisione quando un'istanza viene deallocata.

Spesso, quindi, una variabile impostata per l'autorilascio (*autorelease*) che si utilizzava senza problemi durante l'esecuzione del proprio applicativo, in numerosi metodi nei quali si presupponeva, erroneamente, che non sia stata mai deallocata, a un successiva esecuzione dello stesso potrebbe causare in crash il sistema dopo alcuni secondi, o anche minuti; cosa è accaduto? Nel primo caso il pool non ha effettuato pulizia durante l'utilizzo, mentre nel secondo caso ha provveduto a deallocarla automaticamente.

Quando invochiamo *autorelease* su un oggetto richiediamo il decremento del suo *retainCount* di un'unità in un momento futuro, se questo scenderà a 0 ci sarà il conseguente rilascio automatico. Se andiamo ad analizzare il contenuto del file *main.m*, generato automaticamente in ogni progetto dal wizard, troverete il seguente codice:

```
NSAutoreleasePool * pool = [[NSAutoreleasePool
                                        alloc] init];
int retVal = UIApplicationMain(argc, argv, nil, nil);
[pool release];
```

La prima riga crea e inizializza l'autorelease Pool, che identifichiamo come *Main Autorelease Pool*, cioè autorelease pool principale, successivamente, nella seconda riga, viene avviato il nostro applicativo , e nella terza viene invocato il metodo *release* su tale pool.

Tutto ciò che è contenuto tra la riga di creazione del pool e la chiamata del metodo *release* viene gestito da tale oggetto. È possibile creare uno o più release pool al di fuori del Main Autorelease Pool per rispondere a determinate esigenze? Certo, anche se i contesti in cui fatta questa scelta sono particolari, generalmente comunque ci si affida al Autorelease Pool principale, l'utilizzo più frequente realizzato in maniera manuale da un programmatore, è quello di realizzarne una versione ad

hoc all'interno di un ciclo nel quale vengono generate decine, se non centinaia o addirittura migliaia, di variabili temporanee ma viene sconsigliato in ambiente iPhone perché introduce un sovraccarico computazionale che in alcune situazioni potrebbe obbligare il sistema a terminare la vostra applicazione.

AUTORELEASE POOL

Autorelease Pool provvede in maniera automatica e trasparente a rilasciare la memoria allocata per un oggetto in un non precisato momento durante l'esecuzione del vostro applicativo. Ogni istanza su cui non viene invocato il metodo *alloc* (ma anche , *allocWithZone:*, *copy*, *copyWithZone:*, *mutableCopy*, *mutableCopyWithZone o retain*, viene gestita da tale oggetto automaticamente.

NOTA

RIFERIMENTI WEB

Creazione dell'account, per scaricare l'SDK gratuitamente e consultare la documentazione: http://developer.apple.com/iphone/

Un utilizzo generalmente più comune è quando si utilizzano i thread e si rende necessario creare una pool locale a tale ambiente di esecuzione. L'operazione di pulizia viene generalmente forzata quando le risorse di sistema scendono sotto alcuni limiti, se si verificano tali condizioni inoltre viene invocato per ogni *UIViewController* (e relative classi) il metodo *didReceiveMemoryWarning*, nel quale si dovrebbe provvedere alla rimozione manuale di quelle istanze non strettamente necessarie e non gestite dall'Atorelease Pool. È obbligatorio invocare il metodo *autorelease* su una variabile per affidarla alla gestione dell'Autorelease Pool? No, tale operazione viene effettuata ogni volta che viene creata una variabile senza utilizzare il metodo *alloc*;

```
-(void) inizializzazione {
NSString *newText = [NSString stringWithFormat
                                    @"Lavoro da fare."];
//autorelease automatico, retainCount scenderà da 1
      a  0 in un non precisato istante dopo la fine del
                                              metodo;
//area di utilizzo dellavariabile
}
//al tempo t (random) dopo l'uscita dal metodo verrà
deallocata automaticamente
```

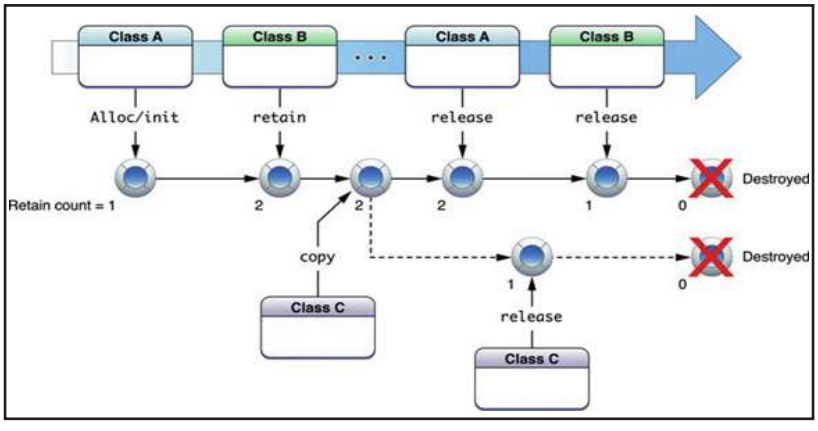

Fig. 2: Un esempio fornito dalla stessa Apple sul funzionamento dell'Autorelease Pool

Questo tipo di inizializzazione, fornito da *NSString*, utilizzando uno dei suoi cosiddetto *Factory Methods* (metodi statici che generano su richiesta istanze della classe), permette di utilizzare la variabile all'interno del metodo in cui questa viene creata e verrà rilasciata, in un non precisato momento, quando tale metodo terminerà, ciò avviene perché quando inizializziamo *newText* questa assume *retainCount* 1 e, alla prossima analisi da parte dell'Autorelease Pool, verrà deallocata poiché, essendo *autorelease*, subirà un decremento automatico di tale valore di 1. Questo utilizzo generalmente non causa problemi, perché la deallocazione automatica da parte dell'autorelease Pool avviene sempre tra una chiamata e l'altra di un metodo; è un approccio comunemente utilizzato quando si utilizzano cicli (*while* e *for* ad esempio) in cui si creano numerose variabili, ma si desidera che vengano rimosse automaticamente quando non più necessarie (in alcune situazioni non è la soluzione migliore in termini prestazionali, quindi è necessario utilizzare la modalità "manuale" adoperando quindi *alloc/retain/release*. Se definissimo una variabile esternamente al metodo, per poi inizializzarla al suo interno per un utilizzo in uno o altri metodi in istanti successivi, verrebbe gestita tale situazione dall'Autorelease Pool?

```
NSString *newText;
-(void)inizializzazione {
newText = [NSString stringWithFormat
                                    @"Lavoro da fare."];
}
-(void)mostraVariabile{
 NSLog(@"%@",newText);
}
```

Invochiamo il metodo *inizializzazione* per inizializzare la nostra variabile e poi, invochiamo in altri metodi quello chiamato *mostraVariabile*, noteremo che casualmente, otterremo dei crash del sistema: ciò è dovuto al fatto che l'autorelease Pool ha rilasciato la variabile tra due successive chiamate di *mostravariabile* (Fig.3): ecco uno dei motivi più frequenti di crash della applicazioni su iPhone: assumere che una variabile sia sempre disponibile quando creata! Le regola, la ribadiamo, è la seguente: "se nella creazione della variabile non viene usato *alloc* oppure *retain*, come vedremo successivamente, tale variabile verrà rimossa in maniera casuale e senza alcuna prevedibilità", memorizzate bene questo concetto perché altrimenti passerete intere giornate cercando di capire perché il vostro applicativo crasha casualmente.

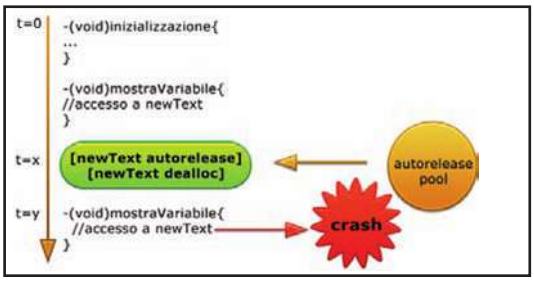

Fig. 3: L'Autorelease distrugge newText tra un'invocazione e un'altra di mostraVariabile, generando un crash

Come evitare questo problema di autorilascio? Un primo modo è quello di utilizzare *retain*, in modo da notificare all'Autorelease Pool che vogliamo utilizzare tale variabile e che siamo in prima persona responsabili del suo rilascio, siamo quindi diventati quindi uno dei co-proprietari di tale variabile; il secondo consiste nel creare una variabile utilizzando *alloc*, metodo che spiegheremo nel prossimo articolo. Questo problema di accesso, dovuto a una scarsa comprensione della gestione in questo contesto, è probabilmente uno dei motivi più frequenti di crash quando si programma per i dispositivi di casa Apple.

NS*

Moltissime classi hanno un prefisso NS, questo è dovuto al fatto che Mac OS X ha radici in NeXTSTEP, (di cui NS rappresentano le iniziali) linguaggio realizzato utilizzando Rhapsody e Mac OS, e che identifica storicamente molte delle classi che fanno parte delle API utilizzate per lo sviluppo su Mac.

SE SI ESAURISCONO LE RISORSE DEL DISPOSITIVO

Nonostante Apple abbia cercato di fornire al singolo applicativo che viene eseguito volontariamente dall'utente (sono ovviamente in esecuzione processi di supporto in maniera trasparente, come quelli di localizzazione, accelerometri e altri sensori, gestione della di telefonia etc) la massima quantità di memoria disponibile, è sempre possibile incorrere nella situazione in cui si occupi una tale quantità di memoria durante l'esecuzione del proprio software, sia a causa dell'utilizzo di grandi strutture dati, sia per errori di programmazione, che generano i cosiddetti *leak* (locazioni di memoria occupate, ma non rilasciabili dall'Autorelease Pool e neppure dall'utente, di cui parleremo nel prossimo articolo), raggiungendo un valore prossimo a 1.5MB, scatenando un sistema di avvisi e invocazioni di metodi automatici; per ogni istanza figlia diretta, o indiretta, di *UIViewController* verrà invocato:

```
- (void)didReceiveMemoryWarning {
    NSLog(@"Memoria insufficiente ricevuto nel
                                UIViewController!!!!");
}
```

mentre per il file chiamato *nome_applicativo AppDelegate.m*, quello responsabile della visualizzazione e della gestione del nostro applicativo :

```
- (void) applicationDidReceiveMemoryWarning:
                            (UIApplication*)application {
    NSLog(@"Memoria insufficiente!!!!");
}
```

Verrà inoltre inviata al nostro applicativo una notifica (argomento non trattato in questa serie di articoli) del tipo *UIApplicationDidReceiveMemoryWarningNotification*. In ognuno di questi metodi dovremo provvedere in maniera manuale alla rimozione di risorse non necessarie al momento, oppure ricreabili quando necessario, come strutture dati ottenute prelevando le proprie informazioni da file disponibili localmente. Nelle due sezioni di codice utilizzate è stato aggiunto un metodo *NSLog*, di stampa a schermo (simile a *printf* del C) allo scopo di consentire in fase di debug di avere un feedback immediato di cosa sta succedendo; è comunque consigliato sempre testare il proprio applicativo sullo smartphone per verificare se il warning di memoria insufficiente viene invocato in qualche situazione. Per forzare questa situazione nel simulatore, che ricordiamo ha accesso alle nostre risorse hardware, parliamo quindi di almeno 1GB di RAM, contro i 128MB dell'iPhone 3G e i 256MB del 3GS, basta, durante una simulazione, cliccare sulla voce "*hardware*" presente nel menu del simulatore e selezionare la voce "*simulate memory warning*". Non bisogna ignorare e neppure tralasciare una corretta implementazione di questi metodi, perché possono essere la soluzione per realizzare progetti di medie-grandi dimensioni dove le richieste di risorse possono superare quelle disponibili.

CONCLUSIONI

In questo articolo, prettamente teorico, abbiamo acquisito ulteriori nozioni, che si riveleranno utili in qualsiasi contesto vi troverete in futuro; nel prossimo articolo continueremo a spiegare come viene gestita la memoria e parleremo di retain, retainCount, release e altri concetti altrettanto importanti . Buona programmazione.

Andrea Leganza

L'AUTORE

Andrea Leganza
Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali e non su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certified Expert, e EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multimediali, anche con iPhone, con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it.

LEAK E ZOMBIE IN AGGUATO...

IN QUESTO ARTICOLO APPROFONDIREMO ULTERIORMENTE I CONCETTI LEGATI ALLA GESTIONE DELLA MEMORIA, IN MODO PARTICOLARE FAREMO LA CONOSCENZA DI QUELLI CHE IN GERGO VENGONO CHIAMATI ZOMBIE E LEAKS

CD WEB
ioProgrammoArt4.zip
cdrom.ioprogrammo.it

Prosegue in questo articolo la trattazione della gestione della memoria. Introdotto nel precedente numero della rivista, l'argomento consentirà una comprensione più approfondita dei vari elementi che abbiamo incontrato nel primo articolo, e che applicheremo, in tutti i futuri articoli dedicati al dispositivo mobile sviluppato dalla casa di Cupertino.

RETAIN E RETAINCOUNT

Come accennato nel numero precedente, esistono due modi per evitare che le proprie variabili vengano rimosse automaticamente dall'Autorelease Pool, utilizzare *retain* o *alloc*.
Nel primo caso basta invocare tale metodo durante la creazione della variabile, dove viene invocato come detto in maniera trasparente il metodo *autorelease*, oppure successivamente, all'interno del metodo in cui questa viene creata.

```
NSMutableString *newText =  [[NSMutableString
                initWithString: @"Lavoro da fare."] retain];
//inizializzata a retainCount = 1, invocato in maniera
trasparente autorelease (-1 all'istante casuale  t=x) e
viene aggiunto 1 utilizzando retain: dopo il passaggio
nell'Autorelease Pool avrà quindi retainCount pari a
1=1(allocazione)-1(autorelease)+1(retain)
```

oppure:

```
NSMutableString *newText =  [NSMutableString
                initWithString: @"Lavoro da fare."];
[newText retain];
```

In questo modo si è creato un legame indissolubile tra la nostra istanza della classe e la variabile che stiamo utilizzando, abbiamo ora il pieno controllo sul suo tempo di vita, se non rilasceremo in futuro tale variabile non sarà mai deallocata. Associata a *retain* esiste la variabile interna a ogni oggetto che abbiamo detto chiamarsi *retainCount*; tale variabile è presente in tutti gli oggetti che discendono da *NSObject* (che come abbiamo detto in un precedente articolo sono la maggior parte); *retainCount* assume un valore numerico intero, è un contatore e indica il numero di oggetti che hanno chiamato il metodo *retain* su un determinato oggetto, hanno quindi dichiarato che la stanno utilizzando, e ne sono quindi co-proprietari: non venite tentati dal desiderio di utilizzare sistemi di stampa a schermo (*NSLog*) per visualizzarlo, perché otterrete spesso valori non coerenti, oppure superiori a quello previsto: per effettuare un'analisi su tale variabile entra in gioco *Instruments*, di cui parleremo in un altro articolo. *RetainCount* assume valore 1 quando la relativa variabile viene inizializzata. Spieghiamo meglio questo che è uno dei concetti fondamentali del linguaggio Objective-C: è normale quando si sviluppa un qualunque applicativo, passare variabili da una classe a una altra invocando i rispettivi metodi, durante questa operazione vengono utilizzati come parametri le istanze delle classi che abbiamo creato in precedenza, poiché però, andando a scavare in profondità parliamo sempre degli amati/odiati puntatori del C, è stato introdotto il concetto di *retain* per evitare che tali variabili vengano deallocate automaticamente se una o più classi hanno dichiarato che la stanno utilizzando; in questo modo si riduce il rischio che avvengano crash in cascata dovuti alla deallocazione automatica effettuata dall'Autorelease Pool (mentre se deallocate manualmente tali variabili tali problemi si possono sempre presentare , e prendono il nome di *Zombie*) che, senza tale artificio, non avrebbe modo di valutare se una variabile è inutilizzata oppure no. Ipotizziamo che una classe *A* invochi un metodo *passaggioStringa* su una classe *B*, passando un parametro di tipo *NSString*, una semplice stringa di testo (per la precisione un oggetto testo, differente sintatticamente dalle stringhe C che sono array di caratteri) quindi, se questa seconda classe desiderasse utilizzare in un altro momento, al di fuori del metodo in questione, tale variabile senza dover effettuare una

REQUISITI

Conoscenze richieste
OOP

Software
MacOS X 10.5.4 o superiore, XCode

Impegno

Tempo di realizzazione

copia profonda (deep copy) del suo contenuto in una variabile locale, potrebbe facilmente effettuare la seguente operazione:

```
//Classe A;
-(void)testRetainCount {
//inivializzazione stringa (è autorelese)
NSMutableString *stringaA = [NSMutableString
            initWithString[@"Telefonare assistenza"]];
   //retainCount =1, essendo autorelase scenderà
  a retainCount=0 in un prossimo futuro, dopo l'uscita
            dal metodo corrente e verrà deallocata;
   //chiamata del metodo di B passando la stringa
  [istanzaB passaggioStringa:stringaA];
  //stringaA verrà deallocata in futuro se non
                              avrà retainCount>0
}

//Classe B
NSMutableString *myString;
-(void)passaggioStringa:(NSMutableString *)stringa{
    myString = stringa; //myString punta alla
            stessa posizione in memoria di stringa
    [stringa retain]; //identico risultato se si
                        effettua[myString retain]
    //stringa ora ha retainCount=2 non verrà deallocata
       perchè verrà decrementata a 1 dall'autorelease pool.
}
```

Poiché *stringaA* è stata creata immediatamente prima dell'invocazione del metodo, avrà *retainCount* pari a 1, che diventerà 0 in futuro poiché è *autorelease*, viene inoltre inserita nell'autorelease pool e per tale motivo verrebbe rilasciata al termine del metodo *testRetainCount,* ma ciò non avviene poiché all'interno di *passaggioStringa* tale valore viene incrementato a 2, utilizzando *retain*; la prima riga effettua una semplice copia tra puntatori, non viene quindi creata alcuna nuova variabile, ma *myString* punta alla stessa locazione di memoria di stringa, e non viene incrementato il *retainCount* di *stringa*, mentre la seconda riga incrementa di 1 il *retainCount* della variabile *stringa,* portandolo al valore 2; in questo modo si evita che questa venga deallocata automaticamente dall'Autorelease Pool al termine del metodo *testretainCount*, infatti fuori da tale metodo avrà valore 1 (dopo che autorelease avrà svolto il suo compito). Effettuare il retain su stringa, oppure su *myString* è ininfluente poiché puntano allo stesso oggetto in memoria. Se non utilizzassimo retain all'interno di *passaggioStringa*:

```
//Classe A;
-(void)testRetainCount {
   NSMutableString *stringaA = [NSMutableString
            initWithString[@"Telefonare assistenza"]];
//chiamata del metodo di B passando la stringa
[istanzaB passaggioStringa:stringaA];
}
//Classe B
NSString *myString;
-(void)passaggioStringa:( NSMutableString *)stringa{
  myString = stringa;
}
-(void)letturaMyString {NSLog (@"%@",myString)}
```

All'uscita del metodo *passaggioStringa* la variabile *stringaA* verrebbe deallocata dopo l'uscita dal metodo *testReatinCount* e ogni accesso successivo in A, e in altri metodi di B, a *myString* (*myString* e *stringaA* sono sinonimi come spiegato) genererebbe un crash del sistema.
La lezione da tenere bene a mente è quindi quella di cercare di avere una chiara visione del tempo di vita delle variabili per evitare ore di debugging.

RELEASE DI UN OGGETTO

Quando decidiamo di notificare all'autorelease pool che non vogliamo più utilizzare una variabile, non vogliamo quindi esserne proprietari (come è stato detto è più opportuno dire coproprietari con altre istanze), invochiamo su di essa il metodo release, che decrementa di un'unità il suo retainCount: utilizziamo il comando *[nome-variabile release],* in questo modo, se la nostra variabile aveva retain count pari a 1 scenderà a 0 e verrà rilasciata immediatamente: attenzione, quindi, se viene deallocata non potrete più accedervi, e, nel caso lo faceste, causerete un crash del sistema (questa operazione errata di accesso viene identificata con l'accesso a uno Zombie, e ne parleremo verso la fine di questo articolo).
Le regola per evitare di incorrere in decine di problemi di gestione della memoria è quella di bilanciare il numero di retain con quello di release, in tal modo, quando si effettuerà l'ultimo release, si farà sempre scendere il *retainCount* a 0 e l'Autorelease pool invocherà il metodo *dealloc()* sulla variabile (il metodo che svolge la funzione di distruttore, quindi responsabile della rimozione delle risorse utilizzat: variabili, strutture dati e canali di comuncazione in primis); ovviamente, se si desidera mantenere in vita un'istanza per tutto il tempo di vita della propria applicazione tale numero dovrà essere sempre maggiore di 0.
Un esempio pratico della variazione della variabile *retainCount* è quello di pensare al numero di scalini necessari per raggiungere un piano, il numero di passi da effettuare per salire sulla sommità prima e poi riscendere è sempre uguale.

Invece di *release*, che effettua il decremento immediatamente, si può anche invocare manualmente *autorelease*, che provvederà a decrementare *retainCount* in futuro: questo approccio non è consigliabile ed è meglio rila-

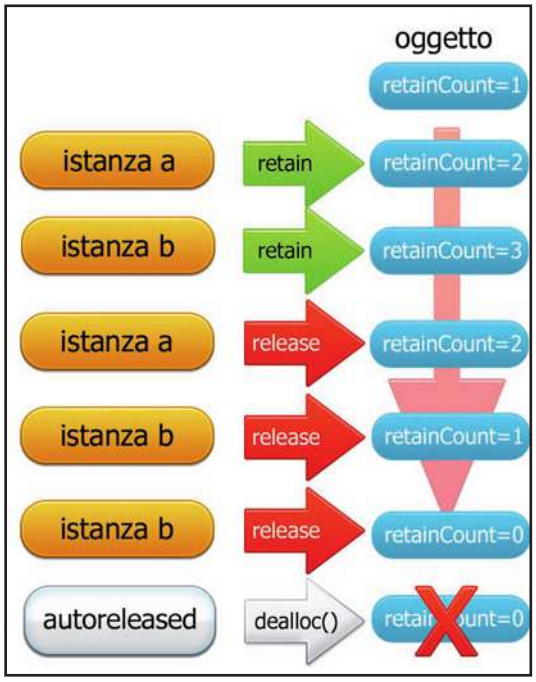

Fig. 1: Un tipico ciclo di vita di una variabile, quando il retaincount raggiunge il valore 0 viene automaticamente rilasciata; si noti inoltre il bilanciamento dei retain con quello dei release, che permettono di farlo scendere a 0

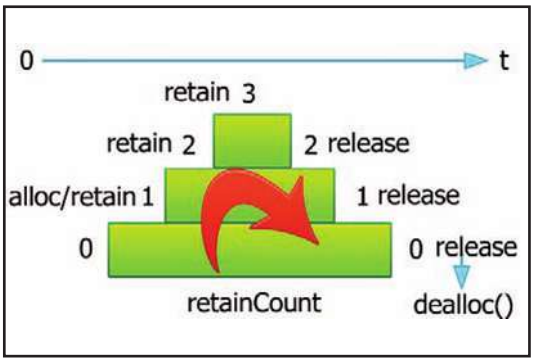

Fig. 2: Durante il tempo di vita di una variabile il retainCount sale e scende, fino generalmente a terminare in un determinato istante t

sciare subito una variabile, sia per liberare il prima possibile una risorsa (e questa è una regola che si dovrebbe applicare in ogni linguaggio di programmazione, ma nel contesto mobile è di fondamentale importanza dove le risorse sono più limitate), ma anche per non incorrere nei problemi di accesso precedentemente spiegati, in primis a variabili deallocate automaticamente in un momento non precisato.
È possibile invocare numerose volte release? In sequenza certamente, ma non è consigliabile se lo si effettua per risolvere problemi di gestione non corretta della memoria, ad esempio quando si dealloca un oggetto e non si effettua il release nel suo distruttore sulle variabili che questa utilizzava e si bypassa il problema in questo modo: generalmente rispecchia una cattiva comprensione del concetto di retain/release, può comunque risultare necessario in particolari situazioni.
Molti metodi effettuano un retain automatico, ad esempio quando aggiungiamo una variabile a un array, e dal punto di vista pratico risulta un'operazione automatica molto utile, poiché gli oggetti inseriti vengono segnalati automaticamente di proprietà di un determinato oggetto e non verranno rilasciati se non quando l'array verrà rilasciato; questa operazione automatica può risultare però causa di problemi quali *leaks* e *zombie* (spiegati in questo stesso articolo). Se si vuole deallocare immediatamente una variabile, disinteressandosi del *retainCount*, basterà invocare il metodo *dealloc()* su di essa, questa operazione e però sconsigliata, poiché, ignorando il *retainCount*, si ignora quante altre istanze la stanno al momento utilizzando e si potrebbero avere numerosi crash del sistema (*Zombie*).
Nella documentazione dell'API viene dichiarato se un certo metodo di una classe effettua oppure no il retain / release sull'oggetto passato come parametro, è comunque possibile utilizzare Instruments per verificare tale comportamento.

NOTA

ULTERIORI APPROFONDIMENTI

Per approfondire i concetti mostrati in questo articolo, consultare la guida *chiamata Memory Management Programming Guide for Cocoa* disponibile su sito developer.apple.com o all'interno della documentazione fornita con Xcode

COSA SONO I LEAK

Non effettuare correttamente la gestione delle risorse in memoria porta ai cosiddetti *leaks*.
Con tale termine si identifica quella variabile che non è più accessibile all'interno di alcun metodo di una classe e non vi è quindi alcun modo per invocare su di essa il *release/autorelease/dealloc*: la variabile è quindi irraggiungibile dal codice realizzato dall'utente e, essendo gestita dall'utente, non può essere rimossa automaticamente dall'Autorelease Pool; in questo modo avremo un dispendio di memoria, in genere incrementale, che potrebbe portare, quando si parla di un numero di variabili medio grande (a seconda della loro singola occupazione di memoria), alla chiusura forzata del nostro programma. Un leak è quasi sempre, quando non è un bug di cui sono responsabili i tecnici di casa Apple (avviene ad esempio quando si utilizza il parser NSXML), segnale di una programmazione errata, disattenta o poco consapevole. Per monitorare tali situazioni si utilizza il tool chiamato *Instruments*, che fornisce in real-time dettagli sul numero e su

quale sequenza di chiamate/comandi hanno generato tale leak. Un esempio di leak è stato descritto nel paragrafo precedente, quando abbiamo parlato del mancato rilascio delle variabili inserite in un array; in tal caso, se non avremo più accesso diretto all'array, non riusciremo più a liberare gli oggetti inseriti al suo interno, che diverranno quindi leak.

IL METODO ALLOC

Il secondo metodo per evitare che una variabile venga deallocata automaticamente, è quello di crearla utilizzando il metodo *alloc*, che imposta il suo *retainCount* a 1 (effettua quindi un *retain* automatico), e sarà quindi nostra cura decrementarlo utilizzando release/autorelease per consentire, poi, nel caso tale valore raggiunga valore 0, all'Autorelease Pool di invocare su di esso il metodo *dealloc*. Quando si usa *alloc,* si richiede che venga allocato lo spazio in memoria necessario per contenere tale istanza. Per inizializzare il suo contenuto si invoca generalmente un metodo che inizia con il prefisso *init*, che completa, quindi, la fase di impostazione del suo stato interno. La mancata invocazione di un metodo *init* su un'istanza, la rende inutilizzabile nella pratica; poiché nel linguaggio Objective-C non esiste una regola per identificare un costruttore, e non è neppure obbligatorio crearlo, anche se risulterebbe di scarsa utilità non realizzarne almeno uno, si ha la possibilità di creare un numero indefinito di costruttori che soddisfino le proprie necessità. Generalmente, comunque, viene utilizzato il prefisso *init* per ognuno di questi, quindi basterà digitare *init* quando comparirà la lista di autocomplete per verificare quali sono stati resi disponibili. Il costruttore più semplice è caratterizzato dalla seguente signature -*(id) init()*: di questo ne parleremo approfonditamente in un prossimo articolo.

```
NSMutableString *stringa;
stringa = [[NSMutableString alloc] initWithString:
                                    @”testo”]; //retainCount=1
```

In questo caso abbiamo creato un'istanza di un oggetto appartenente alla classe *NSMutableString*, in grado di contenere una stringa di dimensione variabile, prima definendola, poi allocandola e infine inizializzando il suo valore interno con una stringa; il *retainCount* è pari a 1 e potremo utilizzarla in qualunque metodo noi desideriamo, poiché il contatore non verrà mai decrementato automaticamente. Quando sarà necessario, potremo invocare su di essa il metodo *release* e farla deallocare, è quindi sempre valida la regola del rapporto uno a uno tra *retain* e *release*. Attenzione, se la variabile viene dichiarata e definita in un metodo, è quindi locale allo stesso, così come nel caso che segue:

```
- (void) initMyString {
    NSMutableString *stringa;
    stringa= [[ NSMutableString alloc]
                                    initWithString:@"testo"];
    [stringa release];
}
```

se non si provvede a rilasciarla al suo interno, prima della fine del metodo, avremo un leak, poiché fuori da questo non sarà più possibile accedervi e rilasciarne la memoria.
Un utilizzo che non presenta tale inconveniente è quello in cui la variabile appartiene a una istanza della classe, ed è accessibile in qualunque suo metodo interno:

```
NSMutableString *stringa;
- (void)initMyString {
stringa = [[ NSMutableString alloc] initWithString:@"testo"];
}
-(void) removeString  {[stringa release];
}
```

Poiché abbiamo accennato dell'inserimento di oggetti in un array, viste le conoscenze ora acquisite, si può affrontare il problema in due modi: utilizzando come struttura dati un *NSMutable Array*, ovveroun array la cui dimensione può cambiare dinamicamente, semplicemente aggiungendo/rimuovendo oggetti e, quindi, invocando alcuni metodi. Leggiamo prima un estratto della documentazione associata a tale classe:

If you do not use garbage collection, when you add an object to an array, the object receives a retain message. When an object is removed from a mutable array, it receives a release message. If there are no further references to the object, this means that the object is deallocated.

Poiché, come è stato già detto in precedenza, non siamo in in ambiente con un vero Garbage Collector, ogni oggetto che verrà inserito nell'array riceverà un *retain*, mentre riceverà un *release* quando verrà rimosso, oppure quando si invocherà release sull'array stesso. Simuliamo l'inserimento:

```
NSMutableArray *array = [[NSMutableArray  alloc] init];
for (int i = 0; i < 10; i++) {
    NSMutableString *convenienceString=
        [NSMutableString stringWithString:@"testo"];
```

```
                                    //retainCount 1 e autorelease
    [array addObject:convenienceString];
    //retain automatico del metodo sul parametro
                                    retainCount 2 di convenienceString
}

...autorelease automatico al tempo x: su tutte le
    variabili inserite nell'array assumono retainCount=
                                                    1...

[array release] //array invia release alle proprie
        variabili quindi il loro retainCount scende a 0 e
    vengono deallocate, array viene ovviamente rilasciato
        dopo questa operazione e deallocato se il suo
        retainCount=0 (non effettua un controllo se il
        retainCount degli oggetti al suo interno è >0)
```

Non abbiamo utilizzato *alloc,* quindi tutte le istanze di *convenienceString* vengono rilasciate automaticamente quando rilasceremo l'array (perchè sono autoreleased). *convenienceString* assumerà *retainCount* pari a 2 poiché *addObject*, da come è dichiarato esplicitamente dalla documentazione dell'API, effettua un retain sul parametro passato al metodo *addObject*; essendo di tipo *autorelease* ognuna delle dieci istanze inserite nel ciclo for verrà decrementata a 1 automaticamente, il successivo rilascio dell'array effettuerà un'ulteriore release su ognuno di questi, e il loro *retainCount* verrà portato a 0, sarà perciò invocato il *dealloc* su ogni variabile di tipo *NSMutableString* che abbiamo inserito.

```
NSMutableArray *array = [[NSMutableArray  alloc] init];
for (int i = 0; i < 10; i++) {
  NSMutableString *convenienceString=
          [NSMutableString stringWithString:@"testo"];
  [array addObject:convenienceString];
                  //retaincount di  convenienceString = 2
  [convenienceString release]; // retaincount di
                                    convenienceString = 1
  [array release]; //array invia release agli oggetti
            che contiene, quindi convenienceString assumerà
 retainCount=0 e verrà dellocato; non si avranno quindi leaks
}
```

In questo caso abbiamo utilizzato *alloc,* quindi è necessario rilasciare la variabile per evitare leaks, poiché altrimenti tutte le istanze di *allocedNumber* presenti nell'array avranno *retainCount* pari a 2 e, quando proveremo a rilasciare l'array, il loro *retainCount* scenderà a 1 impedendone il rilascio e generando un leak dieci in questo preciso esempio, (poiché deallochiamo l'array non potremo più accedervi e, conseguentemente, non avremo modo di accedere a queste variabili con *retainCount* pari a 1). Questo secondo approccio ha il pregio di liberare immediatamente le risorse, e risulta quindi molto adatto in situazioni in cui si genera un numero consistente di inserimenti.
Confrontiamo le due soluzioni: nel primo caso avremo un numero di oggetti incrementale, si passerà da *1+1* della prima iterazione a *n+n* quando si arriverà all'ultima, tale numero diventerà poi pari a *n* solo dopo il termine del metodo, quando le istanze allocate riceveranno l'autorelease; nel secondo caso a ogni iterazione avremo in memoria un numero di oggetti pari a quelli inseriti nell'array e a quello che verrà rilasciato alla fine dell'iterazione, quindi da *1+1* a *n+1* (*n* istanze inserite nell'array e l'oggetto allocato di tipo *NSNumber*). Esiste una terza alternativa, che consiste nell'utilizzare un Pool locale a tale metodo che conterrebbe le variabili create nel primo esempio e le rilascerebbe al termine del ciclo, ma non complichiamo ulteriormente. Bisogna comunque tenere quindi a mente certe considerazioni quando si lavora su strutture dati popolate con oggetti di dimensioni non ridotte.

NOTA

SNOW LEOPARD E XCODE 3.2

I possessori di Snow Leopard hanno modo di utilizzare il software di analisi statica chiamato *Clang*, integrato con XCode 3.2. Purtroppo non è installabile su Leopard 10.5. Questo tool fornisce informazioni dettagliate su tutti i punti in cui possono verificarsi Zombie e Leaks: basterà invocare il comando '*Build and Analyze*' presente nel menu *Build*.

ZOMBIE E EXC_BAD_ACCESS

EXC_BAD_ACCESS rappresenta uno di quegli errori che riceverete più frequentemente quando inizierete a creare istanze di classi in Objective-C, questo errore viene generato quando il vostro codice tenterà di accedere ai cosiddetti *Zombie*, sinonimo di istanze di classi non più disponibili, perché deallocate in precedenza. Prendiamo in considerazione come esempio il seguente codice:

```
 NSMutableString *stringa;
-(void) scatenaZombie {
  [self  initMyString]; [self printMyString];
}
- (void) initMyString {
    stringa = [[ NSMutableString alloc]
                                    initWithString:@"testo"];
    [stringa release]; //la variabile viene deallocata
}
-(void) printMyString  {
    NSLog ("%@",stringa); //accesso ad uno zombie
}
```

Inizializzando nel metodo *initMyString* la nostra stringa, e successivamente deallocandola richiamando il metodo *release,* abbiamo posto fine al suo tempo di vita; se però invocheremo successivamente a tale metodo quello chiamato *printMyString* riceveremo un errore da parte del debugger in esecuzione, poiché tale metodo prova ad accedere a un'istanza non più disponi-

bile. Questo errore si presenta anche nel caso in cui si sbilanci il numero di *retain* e *release*, invocando quindi un numero di questi ultimi maggiore del *retainCount* disponibile:

```
-(void) scatenaZombie {
  [self  initMyString];
  [self removeMyString];
}
- (void) initMyString {
    stringa = [[ NSMutableString alloc]
                              initWithString:@"testo"];
    [stringa release]; //la variabile viene deallocata
}
-(void) removeMyString  {
   [stringa release]; //secondo release : accesso ad
                                          uno zombie
}
```

Purtroppo questo tipo di errore non viene corredato da informazioni sufficienti per comprendere in quale porzione di codice avviene il problema, e per risolvere tale mancanza del debugger è necessario impostare un particolare parametro di configurazione all'interno delle opzioni del file eseguibile generato da XCode. Fate doppio clic sulla voce (ha il nome del vostro progetto) presente come riga all'interno nella colonna di sinistra di XCode, *Goups & Files*, sotto la categoria *Executables* e selezionate *Arguments* dal menu in alto; inserite una riga all'interno del box *Variables to be set in the Enviroment* con il seguente valore: *NSZombieEnabled* e impostando come valore *YES*. Dopo aver effettuato questa operazione, invece di un laconico *EXC_BAD_ACCESS*, otterremo una stringa leggermente più interessante.

```
*** -[CFString retain]: message sent to deallocated
                                    instance 0xd21b50
```

Questa riga di errore ci fornisce l'indirizzo in memoria dell'oggetto che ha causato il crash del software, ma non è ancora sufficientemente comodo per identificare velocemente quale istanza genera il problema. Questa impostazione deve essere utilizzata solamente in fase di sviluppo e non in fase di rilascio poiché non rilascia alcuna variabile allo scopo di consentire lo studio delle variabili Zombie! Una soluzione per evitare di dimenticarsi tale impostazione attiva, consiste nell'inserire il seguente codice nel file *NomeApplicazioneDelegate.m* come prima riga del metodo *applicationDidFinishLaunching*:

```
- (void)applicationDidFinishLaunching:(UIApplication
                                        *)application {
if(getenv("NSZombieEnabled")) {
NSLog(@"NSZombieEnabled!!! Disable when
                                compiling for release");}...
```

Ricordatevi sempre di disabilitarla quando compilerete il vostro software per l'Apple Store.
Per avere ulteriori informazioni su tale errore bisogna impostare un breakpoint che permetterà di avviare il debugger con un dettagliato, e più utile, stack delle chiamate. Selezionando la voce del menu *Run->Show->BreakPoints* si presenterà una finestra dove è possibile inserire e visualizzare i breakpoint impostati. Aggiungiamone uno inserendo il seguente testo:

```
-[_NSZombie methodSignatureForSelector:]
```

Impostando un breakpoint su tale metodo, si farà in modo che il debugger interrompa l'esecuzione dell'applicativo prima che questo crashi, fornendo informazioni sullo stack trace e mostrando quale variabile ha generato il crash. Quando si verificherà un accesso a uno Zombie potrete quindi ottenere informazioni dettagliate, ma soprattutto il nome della variabile e identificare il metodo in cui è avvenuto l'errore. Questo sistema non funziona purtroppo con tutte le classi fornite dall'SDK, e in tal caso diventa tutto più complicato. Per ridurre al minimo questo problema di accesso, basta impostare a *nil* il valore della variabile, immediatamente dopo il suo rilascio (dopo il *release* o *autorelease*), in tal modo qualunque invocazione non avrà alcun risultato, ma almeno non farà crashare il software.

CONCLUSIONI

Tutto quello che abbiamo spiegato in questi due articoli si applica a quasi tutte le classi discendenti da *NSObject* (*NSString* ad esempio si comporta in modo diverso, e spiegheremo il perchè nel prossimo articolo), purtroppo perde di validità quando utilizziamo strutture dati e tipi di dati presi direttamente dal linguaggio C, ad esempio creando variabili di tipo *int* o *float*, che in mano a un programmatore accorto possono fare la differenza in termini di prestazioni per un software. In questo articolo, prettamente teorico, abbiamo acquisito ulteriori nozioni, che si riveleranno utili in qualsiasi contesto vi troverete in futuro; nel prossimo numero della rivista torneremo a trattare della popolazione della tabella e modificheremo il codice creato in automatico per realizzarla; mostreremo anche come popolare le relative celle. Buona programmazione.

Andrea Leganza

NOTA

ZOMBIE E LEAKS

Si accede a uno zombie quando si invocano metodi o si richiedono variabili di un'istanza non più disponibile, perché deallocata (autorelease automatico oppure release manuale); un leak, invece, identifica un'istanza non più accessibile da parte del codice, poiché non è stata deallocata quando era invece necessario.

L'AUTORE

Andrea Leganza
Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali e non su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certified Expert, e EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multimediali, anche con iPhone, con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it.

COME POPOLARE UNA UITABLEVIEW

IN QUESTO ARTICOLO POPOLIAMO LA NOSTRA TABELLA CON UN ELENCO DI VOCI OTTENUTE INTERROGANDO UNA SERIE DI STRUTTURE DATI. NELLA FATTISPECIE VEDREMO COME INSERIRE DELLE SEMPLICI RIGHE, CREARE DELLE SEZIONI E NAVIGARE TRA LE STESSE

CD WEB
ioProgrammoArt4.zip
cdrom.ioprogrammo.it

Nei numeri precedenti di questa seconda serie di articoli dedicati alla programmazione dell'iPhone abbiamo momentaneamente abbandonato il nostro progetto, una ToDo List, per trattare alcuni argomenti di fondamentale importanza relativi al linguaggio Objective-C e alla gestione della memoria in un ambiente non governato da un moderno strumento automatico qual è il Garbage Collector.
Riprendiamo ora a trattare della popolazione della UITableView che avevamo creato utilizzando il wizard di XCode e selezionando il progetto di tipo *Navigation-based Application*.

METODI PER POPOLARE UNA TABELLA

Per semplicità riproponiamo il corpo del metodo *tabelView;cellForRowAtIndexPath* che viene generato automaticamente dal wizard:

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView
    *)tableView cellForRowAtIndexPath:(NSIndexPath
  *)indexPath {static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
     dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
            cell = [[[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
             reuseIdentifier:CellIdentifier] autorelease];
    }
    // Configure the cell.
    return cell;
}
```

Poiché abbiamo sufficientemente trattato in precedenza di *alloc*, *autorelease* e *nil*, iniziamo immediatamente a popolare la nostra tabella nel modo più semplice: inserendo "manualmente" una stringa di testo all'interno di tutte le righe della nostra lista di cose da fare. Prima di ciò, è doveroso descrivere molto brevemente la struttura tipica di una cella, almeno riguardo le versioni fornite nell'SDK.
Una cella svolge la funzione di unità atomica di ogni tabella, ma in pratica è un contenitore anch'essa di numerosi componenti: trovano posto, almeno nelle versioni predefinite, due istanze di *UILabel*, una disponibile con l'identificatore di *textLabel*, l'altra come *detailTextLabel*, una *UIView* con il nome *accessoryType*, un *UIImageView* con il nome di *imageView*, e terminiamo con una *UIView* chiamata *backgroundView*; in realtà la lista di componenti disponibili non è terminata, ma generalmente questi sono quelli utilizzati nella maggior parte di applicazioni, a meno di realizzare customizzazioni relativamente complesse; le due *UILabel* svolgono il compito di mostrare il testo associato alla riga, l'*accessoryType* consente di impostare il tipo di icona da posizionare sulla destra della cella per indicare la possibile presenza di un ulteriore livello di dettaglio, la *imageView*, quando viene associata a un'immagine, la mostrerà sul lato sinistro della cella, prima del testo, la *backgroundView* consente di variare in maniera relativamente avanzata l'aspetto della cella.
Nel caso in cui variare le impostazioni di questi componenti non ottenesse i risultati sperati, si potranno aggiungere uno o più oggetti accedendo alla proprietà chiamata *contentView*, anch'essa un *UIView*, che svolgerà per questi nuovi oggetti la funzione di *superview*: in caso di necessità sarà quindi questo componente che dovrete utilizzare per inserire altre *UILabel*, immagini o altro; se invece si richiedesse una completa ristrutturazione della singola cella, sarà necessario ignorare l'inizializzazione utilizzando gli stili predefiniti (quindi il metodo *initWithStyle:reuseIdentifier* utilizzato dal wizard) e invocare la versione minimale dell'inizializzazione (*init*) e inserire e configurare manualmente ogni componente (impostando posizioni, dimensioni e parametri). Per effettuare una customizzazione è anche possibile utilizzare Interface Builder, ma non tratteremo di questo aspetto nel presente articolo.
Per rendersi conto delle potenzialità delle customiz-

REQUISITI

Conoscenze richieste
OOP

Software
MacOS X 10.5.4 o superiore, XCode

Impegno

Tempo di realizzazione

zazioni possibili per le celle basta semplicemente aprire i *settings*/impostazioni disponibili su iPhone. Quasi ogni schermata presenta una diversa versione di cella. Generalmente si impostano le caratteristiche estetiche comuni a tutte le celle, e che per tale motivo resteranno immutate per tutta la vita della tabella, all'interno del blocco di testo in cui queste celle vengono allocate e inizializzate (quindi all'interno del blocco relativo a *if(cell==nil)*); immediatamente dopo di questo si inserirà il codice necessario a customizzare la riga, in base al contenuto relativo alla singola variabile associata.
Modifichiamo il tipo di cella passando dal tipo predefinito *UITableViewCellStyleDefault* a quello *UITableViewCellStyleValue1* per consentirci di inserire un testo nella parte destra della cella.

```
// Customize the appearance of table view cells.
...
if (cell == nil) {
 cell = [[[UITableViewCell alloc]
      initWithStyle:UITableViewCellStyleValue1
      reuseIdentifier:CellIdentifier] autorelease];
...
}
// Configure the cell.
//Oltre questo commento inseriremo il codice per
                              modificare i testi delle celle
```

Ora possiamo impostare il testo che verrà visualizzato per tutte le righe; per fare ciò dovremo accedere a uno dei componenti visuali contenuti nella cella, identificato dal nome *textLabel*, tale componente, come è stato detto, non è altro che una *UILabel* che consente di mostrare a schermo una stringa di testo semplicemente impostando il valore della sua proprietà *text*:

```
...
cell = [[[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleValue1
              reuseIdentifier:CellIdentifier] autorelease];
// Configure the cell.
cell.textLabel.text= @"Riga di testo";
return cell
}
```

Avviamo il simulatore, notiamo però che non verrà visualizzato alcun testo: questo è dovuto al fatto che, come abbiamo spiegato in precedenza, ogni tabella richiede al proprio delegate (in questo caso la nostra classe *RootViewController* generata dal wizard), attraverso l'invocazione su di questo di due metodi *numberOfSectionsInTableView*: e *tableView*: *numberOfRowsInSection*, il numero di sezioni e quello di righe per ogni sezione disponibile:

```
- (NSInteger)numberOfSectionsInTableView:
                              (UITableView *)tableView { return 1;}
// Customize the number of rows in the table view.
- (NSInteger)tableView:(UITableView *)tableView
          numberOfRowsInSection:(NSInteger)section {
    return 0; }
```

Come si può notare, per default il wizard restituisce per ogni sezione presente, una in questo caso, il valore zero (0): ciò impedisce la creazione di una o più celle: modifichiamo con un valore maggiore di zero, ad esempio 15:

```
// Customize the number of rows in the table view.
- (NSInteger)tableView:(UITableView *)tableView
          numberOfRowsInSection:(NSInteger)section
{ return 15; }
```

Salviamo e avviamo il simulatore: finalmente otteniamo un elenco, di scarsa utilità pratica, ma di indubbio valore didattico. Proviamo ora ad aggiungere un testo nella parte destra. Questa operazione la si compie, così come abbiamo fatto per il *cell.textLabel*, accedendo a *cell.detailTextLabel*, che è sempre un'istanza della classe *UILabel*:

Fig. 1: La tabella mostra un numero di righe a seconda del valore ottenuto invocando tableView: numberOfRowsInSection:

```
 ...
cell = [[[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleValue1
              reuseIdentifier:CellIdentifier] autorelease];
cell.textLabel.text= @"Riga di testo";
cell.detailTextLabel.text = @"Testo";
```

Se avessimo utilizzato come style in fase di inizializzazione della cella quello di default (*initWithStyle:UITable*

NOTA

RIFERIMENTI WEB
Creazione dell'account per scaricare l'SDK e consultare la documentazione:
http://developer.apple.com/iphone

ViewCellStyleDefault), avremo ottenuto come risultato la visualizzazione di tale stringa di testo immediatamente sotto *textLabel*, con questo stile, invece, tale *UILabel* è presentata nella modalità che riteniamo più comoda per una rapida consultazione di una todo list. Impostiamo infine il simbolo grafico di dettaglio, *disclosure button* (Fig. 2), identificato con il simbolo di maggiore, per fare ciò dovremo impostare tale proprietà delle celle all'interno del metodo in cui creiamo le istanze delle celle, parliamo quindi sempre di *tabelView;cellForRowAtIndexPath*:

```
/if (cell == nil) {
 cell = [[[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleValue1
            reuseIdentifier:CellIdentifier] autorelease];
 cell.accessoryType =
          UITableViewCellAccessoryDisclosureIndicator;
}
```

Completiamo questa prima versione impostando il titolo che viene mostrato nella Navigation Bar, la barra di colore blue posizionata nella parte più alta della nostra applicazione, per fare ciò decommentiamo il metodo *viewDidLoad* e aggiungiamo la seguente riga:

```
- (void)viewDidLoad {
    [super viewDidLoad];
          self.title = @"ToDo List";
    // Uncomment the following line to display an Edit
   button in the navigation bar for this view controller.
    // self.navigationItem.rightBarButtonItem =
                                    self.editButtonItem;
}
```

Fig. 2: Abbiamo ora ottenuto una struttura base per una tabella

Perchè abbiamo inserito questo semplice comando all'interno del metodo *viewDidLoad*? Questo è uno dei primi metodi che vengono invocati esplicitamente (ancora prima viene invocato il metodo *initWithNibName*) quando viene analizzato il file *xib* e l'engine del telefono si prepara per mostrarlo visivamente: dopo *viewDidLoad* ogni istanza, direttamente o indirettamente legata a *UIViewController*, invocherà in sequenza *viewWillAppear* e *viewDidAppear* (anche questi inseriti automaticamente dal wizard e anche questi da decommentare in caso di necessità); la scelta è caduta su *viewDidLoad* perché è uno dei metodi consigliati dalla stessa documentazione per effettuare al suo interno operazioni necessarie per la configurazione sia visuale che strutturale del viewcontroller. Cambiamo ora il tipo di tabella da *plain*, quindi visualizzata come un continuo elenco in cui il nome delle sezioni viene presentato come un testo su uno sfondo blu scuro, a *grouped*, allo scopo di distanziare meglio i diversi giorni della settimana (senza doverci preoccupare di impostare parametri per gli header e i footer), per fare ciò apriamo il file *rootviewcontroller.xib* con Interface Builder e, dopo aver selezionato la tabella, utilizzando il Property Inspector, selezioniamo *grouped* dalla

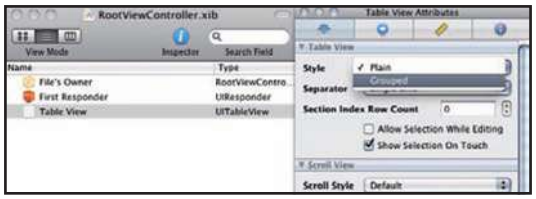

Fig. 3: Impostiamo a grouped il tipo di visualizzazione della tabella

voce a tendina. A questo punto testiamo il comportamento della tabella aumentandone il numero di sezioni, impostiamo a 7, i giorni della settimana che mostreremo nella schermata:

```
- (NSInteger)numberOfSectionsInTableView:
                    (UITableView *)tableView { return 7;}
```

Mostriamo ora il titolo per ognuna delle sette sezioni, il giorno della settimana nel nostro caso; per fare ciò utilizzeremo il metodo richiesto dalla tabella alla nostra classe/delegate, la cui signature è *tableView :titleForHeaderInSection:*; tale metodo riceve come parametro un valore numerico a indicare la sezione che la tabella sta analizzando, un numero nell'intervallo compreso tra da 0 e numerosezioni-1 (7-1 = 6 in questo caso);

```
- (NSString *)tableView:(UITableView *)tableView
          titleForHeaderInSection:(NSInteger)section
{
 switch (section) {
```

NOTA

GLI STILI PREDEFINITI

Gli stili predefiniti per le celle, introdotti nell'SDK 3.0, sono : *UITableViewCellStyleDefault, UITableViewCellStyleValue1, UITableViewCellStyleValue2, UITableViewCellStyleSubtitle*;

```
    case 0: return @"Sunday"; break;
    case 1: return @"Monday"; break;
    case 2: return @"Tuesday"; break;
    case 3: return @"Wednesday"; break;
    case 4: return @"Thursday"; break;
    case 5: return @"Friday"; break;
    case 6: return @"Saturday"; break;
    default: return @""; break; }
    return @"";
}
```

Per ora abbiamo sufficientemente strutturato la nostra interfaccia grafica, dobbiamo ora realizzare un sistema per gestire le nostre informazioni in modo strutturato e facilmente modificabile.

NSARRAY E NSMUTABLEARRAY

Quando si mostra il contenuto di una tabella è necessario accedere in qualche modo al componente Model del pattern MVC, mentre la tabella incarna il View (e il Controller): a seconda delle necessità,si deve scegliere tra una struttura dati di tipo statico, immutabile dopo la sua creazione o dinamico (nella documentazione queste classi contengono generalmente il termine *mutable*). Tra le strutture disponibili vengono spesso adoperate *NSArray* e *NSMutableArray*, facenti parte della libreria *Foundation.framework*; la prima è una classe il cui scopo è quello di contenere un elenco di istanze di altre classi, impedendone però qualunque modifica successivamente alla sua creazione in termine di numero e istanza: non è quindi possibile cancellare oppure inserire nuovi elementi; *NSMutableArray* è semplicemente una versione opportunamente modificata di *NSArray* (è infatti una sua classe direttamente derivata) che consente inserimenti, in praticamente ogni posizione, testa, coda e nel mezzo, e cancellazioni in maniera completamente libera. La scelta di quale struttura sia la più adatta in un determinato contesto è un'operazione generalmente immediata: se il numero di celle può cambiare nel tempo, prevalentemente ad opera dell'interazione dell'utente, la scelta in genere cade su *NSMutableArray* per evitare inutili operazioni di cancellazione e ripopolamento della struttura dati: inserire un nuovo oggetto in un *NSMutableArray* è una singola operazione, mentre nel caso di un *NSArray* comporta una copia profonda della struttura precedente, con un numero di iterazioni quasi sempre pari al numero di elementi presenti (effettuati con un ciclo *for* oppure utilizzando la fast enumeration ad esempio). La rimozione di un elemento da un *NSMutableArray* è anche in questo caso ottenuta con una singola operazione, mentre nel caso dell'*NSArray* ciò comporta sempre effettuare una copia profonda nella quale viene ignorato l'elemento da cancellare. La grande flessibilità di NSMutableArray le impedisce però di essere la soluzione più performante rispetto ad *NSArray*, sia per i meccanismi interni necessari per farla operare, che in termini di occupazione di memoria a causa del codice aggiuntivo inserito al suo interno sotto forma di metodi e altre strutture aggiuntive. Per fortuna convertire una struttura di un tipo all'altro nel caso si sia rivelata la scelta meno adatta è un'operazione relativamente indolore (più leggera da *NSArray* a *NSMutableArray*, viceversa più complessa). Nel nostro caso, per ridurre il numero di righe di codice da digitare, utilizzeremo *NSMutableArray*, per gestire i vari task delle singole giornate, mentre un *NSArray* per raggruppare le informazioni relative ai sette giorni della settimana.

NSDICTIONARY E NSMUTABLEDICTIONARY

Ora che abbiamo definito quale struttura dati conterrà i nostri elenchi di cose da fare, si pone il problema di come rappresentare il singolo task della nostra todo list: analizzando la sua struttura identifichiamo alcune informazioni che identificherebbero il suo stato: una descrizione, che mostreremo nella textLabel e lo stato (*Da fare*, *Fatto*, *Dimenticato*) che verrà utilizzato per le *UILabel* di tipo *descriptionTextLabel*. È possibile utilizzare diverse soluzioni in questo contesto: potremo creare una classe di tipo *NSObject* e al suo interno inserire due campi di tipo *NSString*, oppure delle struct, ma per fare qualcosa di diverso dal solito approccio Object Oriented, adopereremo una classe presente in

Fig. 4: Una serie di sezioni con tre righe per ognuna

NOTA

ULTERIORI APPROFONDIMENTI

Per approfondire questi concetti consultare la guida chiamata Table View Programming Guide for iPhone OS disponibile su sito developer.apple.com o all'interno della documentazione fornita con XCode.

Objective-C disponibile nell'SDK, chiamata *NSMutableDictionary*. La classe *NSMutableDictionary* (e la sua versione "statica", *NSDictionary*) è un'altra delle classi rese disponibili dalla libreria *Foundation.framework*, e permette di creare istanze in cui viene inserito uno o più valori con relative chiavi (ogni coppia viene definita come *entry*), tali chiavi svolgono quindi il compito di identificare univocamente un oggetto tra quelli presenti all'interno dell'*NSDictionary*, consentendone l'accesso in maniera estremamente intuitiva.
Per realizzare un oggetto di tipo *NSMutableDictionary* è necessario inizializzarlo con una serie di chiavi e di valori a queste associati, in corrispondenza uno-a-uno, raggruppati all'interno di due array (di tipo *NSArray*); gli stati possibili saranno "*fatto*", "*da fare*", "*dimenticato*", mentre il testo, ovviamente, varierà a seconda dell'azione da effettuare. Scegliere tra *NSMutableDictionary* e la sua controparte immutabile *NSDictionary* segue lo stesso procedimento decisionale utilizzato per *NSArray*; dal punto di vista delle prestazioni è sicuramente un approccio più pesante rispetto alla realizzazione di una classe derivata da *NSObject*, ma in questo contesto è stato un pretesto per introdurre queste strutture dati e presentare ulteriori classi fornite nell'SDK.

NOTA

FOUNDATION. FRAMEWORK

All'interno di *Foundation.framework* troviamo decine di classi, tra cui *NSDictionary* e *NSMutableDictionary*, *NSArray* e *NSMutableArray*, *NSString* e *NSMutableString*, *NSData* e *NSMutableData*, *NSDate* e *NSError*: è probabilmente la libreria che utilizzerete maggiormente.

```
NSArray *keys  = [NSArray
                arrayWithObjects:@"action",@"status",nil ];
NSArray *values = [NSArray
  arrayWithObjects:@"Chiamare Simone",@"Da Fare",nil];
//Definiamo e inizializziamo il dizionario
NSMutableDictionary *myDict = [[NSMutableDictionary
alloc] initWithObjects:values forKeys:keys]]
```

In questo modo *myDict* conterrà due coppie, una con la chiave "*status*", che assumerà in questo caso il valore "*Da Fare*", mentre l'altra "*Chiamare Simone*" associata alla chiave "*action*". Per accedere ai valori associati alle chiavi presenti in *myDict* basterà utilizzare il metodo *valueForKey*:

```
//otteniamo lo stato associato (fatto, da fare,dimenticato)
NSString * myStatus = [myDict valueForKey:@"status"];
//otteniamo il task associato (@"Chiamare Simone")
NSString * myAction = [myDict valueForKey:@"action"];
```

Ovviamente, un *NSMutableDictionary* (come anche *NSDictionary*) può contenere qualunque oggetto Objective-C e un numero teoricamente infinito di coppie chiave-valore.

POPOLARE LA TABELLA

Per popolare il nostro elenco dovremo prima creare l'array che conterrà le giornate, poiché sarà immutabile utilizzeremo un *NSArray* di sette indici (si ricorda accessibili da *0* a *n-1*), all'interno di ognuna di questi sette indici porremo un *NSMutableArray*, che ci consentirà di variarne il numero di elementi, i nostri task, rappresentati come è stato detto da istanze di *NSMutableDictionary*, nella maniera più veloce possibile. Dichiariamo *giorniSettimana* quale array statico prima del metodo *viewDidLoad*, e procediamo inizializzandolo:

```
NSArray *giorniSettimana;
- (void)viewDidLoad {
 [super viewDidLoad];
 NSLog(@"viewdidload");
 self.title = @"ToDo List";
 giorniSettimana = [[NSArray alloc]
 initWithObjects:[NSMutableArray array],[NSMutableArray
                 array],[NSMutableArray array],[NSMutableArray
                 array],[NSMutableArray array],[NSMutableArray
                                 array],[NSMutableArray array],nil ];
//Inseriamo nel giorno di domenica (indice 0) una
                                                 serie di azioni
[self insertAction:@"Chiamare Simone"
                                 withStatus:@"da fare" inDay:0];
[self insertAction:@"Comprare Latte"
                               withStatus:@"dimenticato" inDay:0];
[self insertAction:@"Registrare Telefilm"
                                 withStatus:@"fatto" inDay:0];
//Inseriamo nel giorno di martedì (indice 2) una serie
                                                 di azioni
[self insertAction:@"Palestra" withStatus:@"da fare"
                                                 inDay:2];
[self insertAction:@"Inviare Email" withStatus:
                                 @"non fare" inDay:2];
[self insertAction:@"Chiamare Gianni"
                                 withStatus:@"dimenticato" inDay:2];
```

Poiché, come abbiamo affermato precedentemente, *NSArray* è una struttura dati immutabile, dovremo impostarne i contenuti in fase di inizializzazione, ciò avviene invocando il metodo *iniWithObjects*. Quest'ultimo accetta un array di oggetti, terminato dal simbolo *nil*: inserendo sette istanze di *NSMutableArray* abbiamo così reso questa struttura dinamica nei contenuti. Ci sarebbero state differenze sostanziali se avessimo utilizzato un *NSMutableArray* per l'elenco dei giorni? Non in questo caso e non in termini di codice, ma sicuramente in termini di prestazioni, quando si inizia a lavorare su strutture dati molto più popolate è buona norma pensare a queste semplici ottimizzazioni che spesso possono rendere meno oneroso in termini di risorse e prestazioni il vostro software.
Come viene affermato dalla documentazione relativa alla classe, *NSArray* mantiene un legame *strong*, forte, con gli oggetti che vengono inseriti al suo interno invocando su di essi un *retain*, e, venendo a conoscenza di questo comportamento non è stato necessario utilizzare la tipica procedura *alloc/init* (*[[NSMutableArray alloc] init]*) nella fase di inserimento, abbiamo infatti utilizzato il *convenience constructor*, un

costruttore che non effettua *alloc/retain* sull'istanza creata, e che per tale motivo porterà all'autorelease dei singoli oggetti, ma, poiché questi riceveranno un *retain* non saranno deallocati. Se avessimo utilizzato *alloc+init*, ogni oggetto avrebbe avuto un *retainCount* pari a due, uno per l'inizializzazione e uno dovuto all'inserimento all'interno dell'array, impedendo quindi la completa deallocazione dell'array quando su di questo richiameremo il metodo *release* (gli oggetti al suo interno avranno un *retainCount* pari a uno invece che zero, come richiesto per la completa deallocazione della struttura dati). Ora che abbiamo creato un array di sette indici, ognuno contenente un *NSMutableArray*, realizziamo un semplice metodo per inserire all'interno dei giorni i task (che rappresentiamo con *NSMutableDictionary*):

```
(void)insertAction:(NSString *)action withStatus:
                        (NSString *)status inDay:(int)day {
NSArray *values = [NSArray
                        arrayWithObjects:action,status,nil];
NSArray *keys = [NSArray
                    arrayWithObjects:@"action",@"status",nil ];
[((NSMutableArray *)[giorniSettimana objectAtIndex:
                day]) addObject:[[NSMutableDictionary alloc]
                        initWithObjects:values forKeys:keys]];
}
```

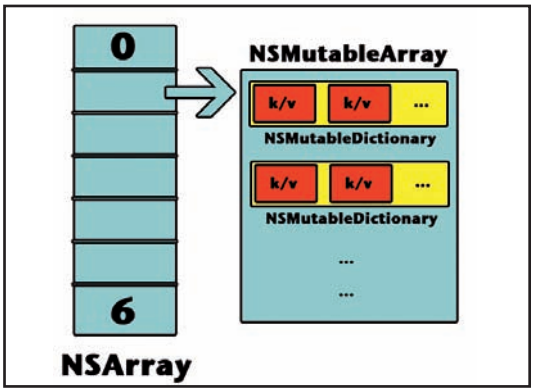

Fig. 5: Ecco come sono state strutturate le informazioni che vogliamo gestire

Non allarmatevi se riceverete un warning che informa della mancata presenza del metodo, basterà inserire la sua signature (*-(void)insertAction:(NSString *)action withStatus:(NSString *)status inDay:(int)day)* seguita da un punto e virgola prima della riga *@end* all'interno del file *rootViewcontroller.h*. Poiché il numero di utilizzi di questo metodo è ridotto, si è scelto di non ottimizzare in alcun modo l'utilizzo dell'array *keys*, essendo una struttura che non viene in alcun modo modificata durante il tempo di vita della nostra applicazione, potrebbe infatti essere sostituita utilizzando la direttiva *#define*, evitando quindi qualunque inizializzazione di istanze (in questo caso una per ogni invocazione del metodo). Il comportamento del metodo è abbastanza intuitivo: vengono passate due stringhe che contengono il task e il suo stato, infine l'indice del giorno da utilizzare (anche in questo caso si potevano usare i *#define* per rendere il codice più leggibile all'interno di *viewDidLoad*); viene prelevato l'oggetto di *giorniSettimana* presente in posizione i-esima, effettuato il casting a *NSMutableArray* per consentire al sistema di code completition di mostrarci solo i metodi coerenti con tale classe (infatti *objectAtIndex* restituisce un'istanza di tipo *id* che rende inutilizzabile tale funzionalità, ne avevamo parlato in uno dei primi articoli dedicati alla programmazione iPhone) e inserito al suo interno un *NSMutableDictionary* contenente le due coppie con chiavi *action* e *status*.

AGGIORNAMENTI

Cosa ci resta da fare? Per come abbiamo strutturato i nostri dati identificheremo con le sezioni i singoli giorni della settimana, mentre le righe saranno i singoli *NSMutableDictionary* in esse presenti. Prima dobbiamo modificare il valore restituito da *tableview*: *numberOfRowsInSection*: in modo che fornisca il corretto numero di righe presenti nella sezione/giorno. Basterà invocare il metodo *count* su ogni singolo oggetto del nostro array:

```
- (NSInteger)tableView:(UITableView *)tableView
            numberOfRowsInSection:(NSInteger)section {
    return [((NSMutableArray *)[giorniSettimana
                        objectAtIndex:section]) count];  }
```

Per popolare la tabella basterà ottenere le informazioni *action/status* per ogni *NSMutableDictionary* presente nelle varie sezioni/giorni:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
     cellForRowAtIndexPath:(NSIndexPath *)indexPath {...
NSString *action = [(NSMutableDictionary
                    *)[((NSMutableArray *)[giorniSettimana
                        objectAtIndex:indexPath.section])
    objectAtIndex:indexPath.row] valueForKey:@"action"];
NSString *status = [(NSMutableDictionary *)[((NSMuta
                        bleArray *)[giorniSettimana
                        objectAtIndex:indexPath.section])
    objectAtIndex:indexPath.row] valueForKey:@"status"];
cell.textLabel.text = action;
cell.detailTextLabel.text = status;
return cell; }
```

CONCLUSIONI

Nel prossimo articolo introdurremo altri concetti e tecniche per utilizzare la tabella, parleremo di cancellazioni, inserimenti e altre operazioni.

Andrea Leganza

L'AUTORE

Andrea Leganza
Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali su piattaforme e con linguaggi eterogenei. Certificato Adobe ACE - Adobe Flex 3 and AIR Certified Expert, e EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto 2° Livello FIN, è attualmente impegnato in numerosi progetti multimediali, anche con iPhone, con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it.

GESTIONE DELLE CELLE NELLE TABELLE

IN QUESTO ARTICOLO TRATTIAMO DELLE VARIE FUNZIONALITÀ OFFERTE DALL'SDK PER INSERIRE E CANCELLARE UNA O PIÙ RIGHE NELLE TABELLE: ELEMENTO FONDAMENTALE NELLA COSTRUZIONE DELLE INTERFACCE, SIA PER L'INPUT CHE PER LA VISUALIZZAZIONE DEI DATI

CD WEB
iphone_tabella146.zip
cdrom.ioprogrammo.it

Nell'articolo precedente abbiamo prima mostrato come si popola una tabella utilizzando una struttura statica, un array, e successivamente abbiamo realizzato un metodo per popolarne una utilizzando una struttura dinamica (di tipo *NSMutableArray*):

```
-(void)insertAction:(NSString *)action
        withStatus:(NSString *)status inDay:(int)day {
NSArray *values     = [NSArray
                       arrayWithObjects:action,status,nil];
NSArray *keys       = [NSArray
                 arrayWithObjects:@"action",@"status",nil ];
[((NSMutableArray *)[giorniSettimana object
          AtIndex:day]) addObject:[[NSMutableDictionary
              alloc] initWithObjects:values forKeys:keys]];
}
```

Poiché il metodo che abbiamo mostrato non è invocabile, in risposta alla pressione di un pulsante o di un altro componente, non essendo di tipo *IBAction* (che ricordiamo essere un tipo di metodo che non accetta altri parametri se non il componente grafico che lo ha invocato), dobbiamo percorrere un'altra strada per poter consentire con facilità l'esecuzione di una qualunque modifica in risposta all'interazione dell'utente: esistono per nostra fortuna svariate tecniche per effettuare ciò, una di queste consiste nel modificare la struttura della tabella fornendo campi di testo e pulsanti necessari per modificare i contenuti della struttura dati che utilizziamo, e un'altra invece si realizza presentando un ulteriore *UIViewController* con relativa *UIView*, potremo anche presentare un *AlertView* customizzato; utilizzeremo questa terza procedura per aggiungere un nuovo evento.

REQUISITI

Conoscenze richieste
OOP

Software
MacOS X 10.5.4 o superiore, XCode

Impegno

Tempo di realizzazione

LA NAVIGAZIONE TRA "SCHEDE"

L'utilizzo dell'iPhone è principalmente basato sul concetto di consultazione progressiva di informazioni, come è stato spiegato quando abbiamo trattato dei passaggi necessari per arrivare alla scheda di dettaglio di una serie di tabelle. La navigazione è una funzionalità fornita dal componente che contiene la nostra tabella realizzata nel file *RootViewController.h/.m* e nel relativo file *.xib*, di tipo *UINavigationController*: se andiamo infatti ad esplorare le variabili presenti nel file *nomeprogettoAppDelegate.h* possiamo notare che ne è presente una di tipo *UINavigationController*, di tipo *IBOutlet*, chiamata *navigationController*, che nel corpo del metodo *applicationDidFinishLaunching*, è utilizzata dal contenitore principale, la variabile *window*, come fonte della view da mostrare a schermo. In pratica, cosa viene realizzato automaticamente quando si crea un progetto di questo tipo navigation-based? La *UIWindow* contiene al suo interno una istanza di un *UINavigationController*, che a sua volta include una tabella; la presenza del navigation controller ci permette con estrema facilità di gestire inserimenti, e ovviamente rimozioni, di un numero virtualmente illimitato di ulteriori *UIViewController* (nel cui interno sono presenti quindi *UIView*): in questo modo è possibile realizzare un complesso sistema di navigazione, che potremo dire "a schede" (con il quale intendiamo il binomio *UIViewController* e relativo *UIView*). Riguardo l'*UINavigationController*, oltre a gestire la navigazione, consente di inserire dei pulsanti nella sua barra superiore, chiamata *navigationBar*. Parleremo di questi aspetti successivamente, ma era necessario presentare almeno queste minime informazioni perché l'*UINavigationController* verrà utilizzato quando dovremo implementare i pulsanti per effettuare le cancellazioni e gli inserimenti degli eventi.

NSSTRING

Prima di iniziare è doveroso trattare delle stringhe. In uno degli articoli precedenti, dove avevamo trattato della gestione della memoria, e

degli utilizzi di retain/release avevamo evidenziato che le stringhe del tipo *NSString* non erano soggette a queste regole, avevamo infatti utilizzato negli esempi istanze di *NSMutableString*; è ormai necessario aprire una parentesi e spiegare perché *NSString* si esenta dalla gestione delle memoria dinamica, soprattutto perché è uno dei tipi di dato maggiormente utilizzati negli applicativi iPhone, dove spesso si richiede all'utente di compilare uno o più campi di testo. La spiegazione è di estrema facilità: ogni istanza di *NSString* è sempre gestita come una costante rappresentata in Unicode, e non vi è quindi modo di modificare il contenuto di una variabile di questo tipo senza che venga, esplicitamente o implicitamente, sostituita con una nuova; se andassimo ad analizzare come vengono gestite due variabili in memoria il cui testo è identico, ad esempio "prova", scopriremo che queste due puntano alla stessa locazione di memoria, indifferentemente dal momento in cui le abbiamo istanziate, e alla classe in cui tale operazione avviene: lo scopo di tale comportamento è ovviamente quello di ottimizzare il consumo di memoria, infatti in questo modo si possono avere migliaia di alias di un solo testo (n alias diversi e una sola variabile in memoria), e non migliaia di istanze con contenuto identico (*n* variabili in memoria) la cui occupazione è quindi di tipo lineare. Quando, durante la fase di compilazione, verranno analizzati i sorgenti, verrà avviato un processo che come risultato farà in modo che tutte le variabili con stesso contenuto punteranno ad una sola locazione di memoria. Essendo quindi le stringhe delle costanti vengono esentate dalla necessità di *retain/release/autorelease* e pertanto non ci si deve mai preoccupare di invocare tali metodi per gestirne la memoria. Ultima nota riguardante le stringhe: queste consentono di utilizzare il simbolo di doppia uguaglianza per verificare se i loro caratteri sono identici, anche se l'operatore == non ha subito overloading, questo è dovuto al fatto che tale operatore verifica, per default, se i due indirizzi puntati da due variabili sono identici, e come appena spiegato ciò è sempre vero se due stringhe hanno uguali caratteri.

```
NSString *myString = @"prova";
NSString *myString2 = @"prova"; //punteranno in
                                memoria alla stessa locazione
if (myString==myString2) {codice sempre
                                eseguito}
```

Ciò non è vero quando utilizziamo due *NSMutableString*, in questo caso l'uguaglianza non avrà l'esito sperato perché le due variabili puntano sempre a locazioni diverse; in questo caso è quindi necessario utilizzare il metodo *isEqualTo String*.

```
NSMutableString *myString = [NSMutableString
                        stringWithString:@"prova"];
NSMutableString * myString2 = [NSMutableString
                        stringWithString:@"prova"];

if (myString == myString2) NSLog(@"sono uguali");
                //non viene mai verificata come condizione
if ([myString isEqualToString: myString2])
                NSLog(@"sono uguali con equaltostring");
```

Anche se è quindi lecito utilizzare l'== con le stringhe di tipo *NSString* si rischia, in caso di dimenticanza, di utilizzarlo anche con le *NSMutableString*, o peggio, con istanze di altri oggetti: il consiglio è quindi di usare *isEqualtoString* per evitare possibili errori.
Poiché dovremo prelevare del testo da uno o più campi, non avere questa consapevolezza potrebbe creare problemi inizialmente a quei lettori meno attenti/preparati, inducendoli a passare ore cercando di deallocare queste variabili senza successo ottenendo crash continui.

NOTA

NSSTRING

Ogni stringa di tipo NSString è una costante, quindi non è soggetta ad alcuna gestione da parte dell'utente e neppure dal sistema di gestione automatica delle risorse; inoltre tutte quelle stringhe che hanno identico valore puntano ad una stessa locazione di memoria.

I POSSIBILI STATI DELLE CELLE DI UNA TABELLA

Ogni cella di una tabella si comporta come un automa a stati finiti, può passare quindi tra un numero limitato di diverse configurazioni, ognuna delle quali generalmente condiziona il suo aspetto. Successivamente, se coerente con il comportamento atteso, anche il contenuto della relativa "riga" associata nella struttura dati

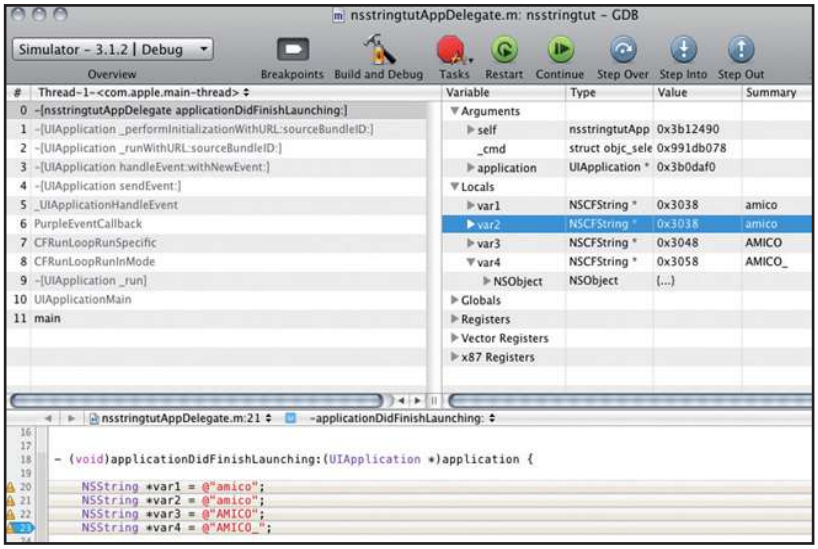

Fig. 1: Due stringhe con ugual contenuto di testo punteranno alla stessa locazione in memoria. Nel nostro esempio: var1 e var2

collegata: abbiamo uno stato "normale" durante il quale si effettuano le semplici operazioni di scorrimento e di selezione delle "righe" (*UITableViewCellEditingStyleNone*), poi uno stato di "modifica" che può essere di cancellazione, oppure di inserimento (esiste anche un terzo, quello di riordino ma che non viene utilizzato in questo modo), il primo prende il nome di *UITableViewCellEditingStyleDelete*, mentre il secondo *UITableViewCellEditingStyleInsert*: tali "stati" sono quindi degli indicatori per consentire di valutare se una determinata cella è al momento, visivamente parlando, in uno stato di modifica oppure no; esiste inoltre una proprietà della cella, editing, di tipo booleano, il cui scopo è indicare l'effettivo stato della cella, teoricamente sarebbe infatti possibile impostare una cella con uno stile visivo di modifica e renderla invece non editabile; generalmente *editing* non viene consultata perché ci si affida al fatto che una cella con un determinato stile ha già automaticamente impostato tale valore a *true* o *false*; se si utilizzano i metodi relativi alla tabella, quelli che vengono generati automaticamente dal wizard, non avrete necessità di valutarla. Infatti noteremo nel prossimo paragrafo che semplicemente verrà consultato lo stile visuale e non editing. Generalmente lo stato di cancellazione si evidenzia in due modi: o si presenta con un pulsante rosso, simile ad un segnale di stop, sul lato sinistro della cella che, quando premuto, farà comparire un bottone sul lato destro della cella per confermare tale operazione, oppure con solamente un bottone sulla destra; questo secondo aspetto si presenta quando si effettua la cancellazione effettuando lo scorrimento del dito da destra verso sinistra sulla cella desiderata. Molti metodi da utilizzare per gestire tali operazioni vengono generati automaticamente dal wizard del progetto, basterà decommentarli e personalizzarli a seconda delle proprie necessità.

NOTA

UGUAGLIANZA TRA STRINGHE

L'utilizzo di == quale operatore per verificare l'uguaglianza tra due stringhe di tipo NSString funziona, nonostante non sia soggetto ad overloading, poiché verifica se i due indirizzi sono identici. Situazione vera quando due stringhe puntano ad una stessa locazione di memoria, e ciò avviene perché stringhe con contenuti uguali puntano alla stessa memoria.

GESTIRE LE CANCELLAZIONI

Gestire la cancellazione di una o più celle, quando è necessario rispondere all'interazione con l'utente, si rivela per nostra fortuna, un'operazione estremamente semplice; per quanto riguarda l'utilizzatore dell'interfaccia, tale operazione viene invocata automaticamente quando si scorre il dito sulla cella con un movimento da destra verso sinistra, oppure premendo un pulsante, generalmente presente in alto a destra, con label *edit*, che mostrerà successivamente un pulsante rosso sul lato sinistro di ogni riga; dal punto di vista del programmatore, per mostrare il pulsante di edit in alto a destra, basterà inserire la seguente riga all'interno del metodo *viewDidLoad*, ovviamente dopo la chiamata al metodo della classe padre:

```
[super viewDidLoad];
self.navigationItem.rightBarButtonItem =
                                    self.editButtonItem;
```

In questo modo si è impostato il pulsante destro presente nella *navigationBar* del *UINavigationController* con uno predefinito, presente in tutti gli *UIViewController*, il cui comportamento, preimpostato, è quello di invocare il metodo *(void)setEditing:(BOOL) editinganimated:(BOOL)animated* (come esplicitamente descritto nelle documentazione di *UIViewController*), per la *UITableView* questo metodo è stato già implementato internamente alla classe e non sarà necessario crearlo o modificarlo: provvederà ad iterare per tutte le righe e mostrare il pulsante rosso sul lato sinistro di ognuna di queste (se è abilitata alla cancellazione). Il pulsante è, per informazione, appartenente alla classe *UIBarButtonItem* e, per supportare la creazione di una cella nel prossimo paragrafo, la utilizzeremo per istanziare un bottone ad hoc (quello nella parte sinistra). Abbiamo ora reso disponibili due modalità di cancellazione, quella multipla, il cui utilizzo termina con l'ulteriore pressione del pulsante presente a destra nella navigationBar, che assume il testo "done" successivamente alla prima interazione, e quella singola, scorrimento destra-sinistra del dito sulla cella.
Ora è necessario abilitare la cancellazione di una o più celle: cosa che si realizza implementando un metodo per effettuare una o più cancellazioni. Per supportare questa operazione, basterà decommentare, il metodo che riportiamo di seguito:

```
-(void)tableView:(UITableView*)tableView
   commitEditingStyle:(UITableViewCellEditingStyle)
      editingStyle forRowAtIndexPath:(NSIndexPath
                                    *)indexPath {

if (editingStyle ==
                    UITableViewCellEditingStyleDelete)
{
 // Delete the row from the data source.
 //[tableView deleteRowsAtIndexPaths:[NSArray
                        arrayWithObject:indexPath]
withRowAnimation:UITableViewRowAnimationFade];
```

```
    }
}
```

Decommentando questo metodo si rendono cancellabili tutte le celle; se si volesse creare la logica per proteggerne una o più si dovrà decommentare il seguente metodo, sempre generato dal wizard del progetto, e decidere, a seconda della cella ricevuta, se restituire *YES* oppure *NO*:

```
- (BOOL)tableView:(UITableView *)tableView
  canEditRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Return NO if you do not want the specified
                                item to be editable.
    return YES;
}
```

Tornando al metodo utilizzato per modificare la tabella, analizzando il comportamento di default fornito dagli ingegneri Apple, si noterà che viene prima verificato lo stile attuale della cella sulla quale tale metodo è stato invocato e, se corrisponde a *UITableViewCellEditingStyleDelete*, si può provvedere a cancellare la "riga" presente nella nostra tabella. La parte di codice che quindi dovremo modificare è quella precedente alla seguente riga (che dovrete decommentare se fosse commentata):

```
[tableView deleteRowsAtIndexPaths:[NSArray
                              arrayWithObject:indexPath]
withRowAnimation:UITableViewRowAnimationFade];
```

Questa chiamata provvede a cancellare visivamente la cella, infatti viene invocata sulla nostra tabella, *tableView*, ma se non provvederemo a rimuovere anche la relativa entità nella nostra struttura dati avremo un'incoerenza tra model (la struttura dati usata, *NSMutable Array*) e view (le celle della tabella), il nostro software verrà terminato con un crash molto esplicativo questa volta:

Terminating app due to uncaught exception 'NSInternalInconsistencyException', reason: 'Invalid update: invalid number of rows in section 0. The number of rows contained in an existing section after the update (3) must be equal to the number of rows contained in that section before the update (3), plus or minus the number of rows inserted or deleted from that section (0 inserted, 1 deleted).

Basterà quindi aggiungere la seguente riga prima delle cancellazione della cella della tabella:

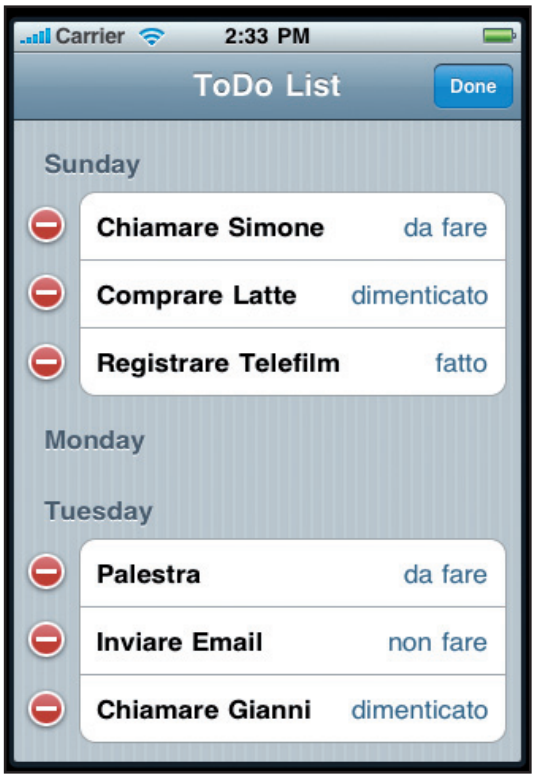

Fig. 2: Ecco come si presenta la modalità di cancellazione multipla

```
//Cancellazione dal MODEL
[((NSMutableArray *)[giorniSettimana
                    objectAtIndex:indexPath.section])
                removeObjectAtIndex:indexPath.row];

//Cancellazione dal VIEW
[tableView deleteRowsAtIndexPaths:[NSArray
                              arrayWithObject:indexPath]
withRowAnimation:UITableViewRowAnimationFade];
```

Come si nota la gestione della cancellazione si è rivelata un'operazione molto semplice: ha comportato solo decommentare il metodo relativo e l'aggiunta di una sola riga per far riflettere questa modifica anche alla struttura dati.

GESTIRE GLI INSERIMENTI

Esistono svariati metodi per aggiungere una "riga" nella nostra tabella, come abbiamo detto in precedenza, in questo tutorial utilizzeremo un *UIAlertView* customizzato; l'*UIAlertView* è una *UIView* che generalmente, in maniera modale, quindi impedendo intera-

NOTA

RIFERIMENTI WEB

Creazione dell'account, per scaricare l'SDK e consultare la documentazione: http://developer.apple.com/iphone/

zione con gli altri componenti, presenta uno o due, bottoni mostrando un messaggio di testo e richiedendo una scelta: un esempio tipico è il messaggio che richiede di autorizzare la consultazione della posizione del telefono per una qualche funzionalità di localizzazione che si trova in numerosi applicativi. Noi lo modificheremo per le nostre necessità utilizzando delle funzioni nascoste, non pubblicamente rese disponibili da Apple, che possono essere utilizzate per inserire un qualunque numero di campi di testo (*UITextField*) al suo interno.

Dovremo effettuare tre operazioni, la prima consiste nell'inserire un pulsante nella *navigationBar* sul lato sinistro che ci consenta di inserire una nuova scheda, presentandoci quindi l'*UIAlertView*; la seconda è implementare il metodo *addItem*, che provvederà a creare l'*UIAlertView*; la terza sarà l'implementazione di un secondo metodo in risposta alla conferma della creazione dell'evento, successivamente alla pressione dell'*UIAlertView*.

Per poter effettuare la prima operazione, quella di visualizzazione del pulsante sulla sinistra, provvediamo aggiungendo le seguenti righe sempre nel metodo *ViewDidLoad*, immediatamente dopo il comando che abbiamo utilizzato per mostrare il pulsante di cancellazione:

```
self.navigationItem.rightBarButtonItem =
                                    self.editButtonItem;

UIBarButtonItem *addButton =
                                    [[UIBarButtonItem
  alloc] initWithBarButtonSystemItem:UIBarButton
                                    SystemItemAdd target:self
                                    action:@selector(addItem:)];

self.navigationItem.leftBarButtonItem =
                                    addButton;

[addButton release];
```

Il codice è molto semplice: si crea un'istanza di un bottone, come una normale istanza di un oggetto, lo si imposta con uno stile visuale che mostra il simbolo di più (+) ad indicare l'inserimento, e si setta come target responsabile della gestione degli eventi il metodo *addItem* nella stessa classe in cui ci troviamo (*self* sta proprio ad indicare di utilizzare questa classe).

Dopo aver associato il pulsante di sinistra, *leftBarButtonItem*, con il nostro bottone, dovremo ricordarci di rilasciare la risorsa, questo è dovuto al fatto che, avendo utilizzato alloc/init, avremo un *retainCount* di 1, l'assegnazione del pulsante di sinistra, come segnalato dalla documentazione, provvede ad effettuare un *retain*.

Quindi, al termine del metodo, se non effettuassimo un *release*, avremo un *retainCount* di 2, che non verrebbe mai deallocato, neppure nel caso rilasciassimo il *UIViewNavigator*, diventando quindi *leak*. Passiamo ora alla seconda fase:

```
- (void)addItem:sender {

UIAlertView *alert = [[[UIAlertView alloc]
   initWithTitle:@"Create new Event" message:@""
                                    delegate:self
cancelButtonTitle:@"Cancel"
otherButtonTitles:@"Submit", nil
                                    ] autorelease];

[alert addTextFieldWithValue:@"Sunday"
                                    label:@"Day of the week"];
[alert addTextFieldWithValue:@"" label:@"What"];
[alert addTextFieldWithValue:@""
label:@"Status"];
[alert show]}
```

L'implementazione del metodo *addItem* crea un *UIAlertView* con titolo "Create new Event", impostando due bottoni, uno di annullamento (con label *Cancel*) e uno di conferma (con label *Submit*); entra ora in gioco l'utilizzo di un metodo non pubblicato da parte di Apple associato alla classe *UIAlertView*, tale metodo *addTextFieldWithValue*, consente di inserire un numero illimitato di campi di testo che consulteremo quando confermeremo la creazione dell'evento.

Per accedere a questi campi di testo dovremo utilizzare un altro metodo nascosto, [*alertView textFieldAtIndex:i*], dove ovviamente i è l'indice, partendo da 0, a cui è associato ogni campo, 0 per il campo del giorno della settimana, 1 per quello dell'azione, 2 per lo stato dell'azione.

Per quanto riguarda la pressione dei due pulsanti disponibili implementiamo il metodo *alertView: clickedButtonAtIndex*, invocato automaticamente quando premeremo *cancel* o *submit*.

```
- (void)alertView:(UIAlertView *)alertView
     clickedButtonAtIndex:(NSInteger)buttonIndex
{
   if (buttonIndex !=
                    [alertView cancelButtonIndex])
{
```

NOTA

UIALERTVIEW

Un componente di tipo *UIAlertView* consente di presentare messaggi, con una o più possibili scelte, in maniera modale, intercettando quindi l'interazione dell'utente.

NOTA

ULTERIORI APPROFONDIMENTI

Per approfondire questi concetti consultare la guida chiamata *Table View Programming Guide for iPhone OS* disponibile su sito developer.apple.com o all'interno della documentazione fornita con XCode.

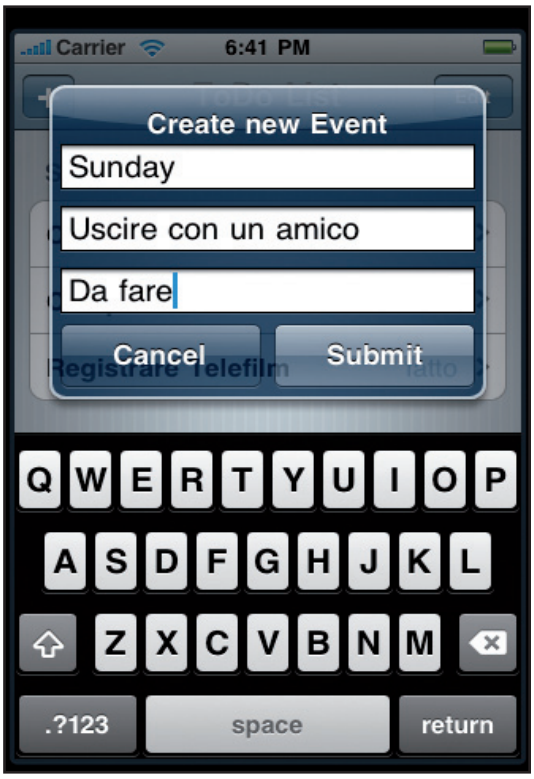

Fig. 3: L'UIAlertView che mostra i campi di input

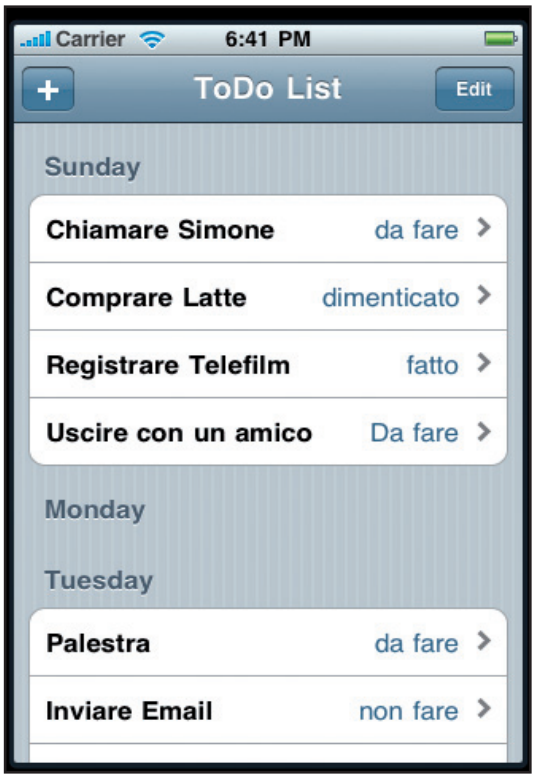

Fig. 4: Il risultato dell'inserimento

```
    short day = 0;
if ([((UITextField *)[alertView textFieldAtIndex:0])
        .text isEqualToString:@"Sunday"]) day = 0;
else if ([((UITextField *)[alertView
                                    textFieldAtIndex
    :0]).text isEqualToString:@"Monday"]) day = 1;
else if ([((UITextField *)[alertView
                                    textFieldAtIndex
    :0]).text isEqualToString:@"Tuesday"]) day = 2;
else if ([((UITextField *)[alertView
                                    textFieldAtIndex
    :0]).text isEqualToString:@"Wednesday"]) day =
                                                3;
else if ([((UITextField *)[alertView
                                    textFieldAtIndex
  :0]).text isEqualToString:@"Thursday"]) day = 4;
else if ([((UITextField *)[alertView
                                    textFieldAtIndex
      :0]).text isEqualToString:@"Friday"]) day = 5;
else if ([((UITextField *)[alertView textFieldAt
  Index:0]).text isEqualToString:@"Saturday"]) day
= 6;
[self insertAction:((UITextField *)[alertView
  textFieldAtIndex:1]).text withStatus:((UITextField
*)[alertView textFieldAtIndex:2]).text inDay:day];

[(UITableView *)self.view reloadData]; //
//oppure [tableView reloadData] se  tableView è il
                        nome dell'IBOutlet della tabella
}
}
```

Come prima operazione verifichiamo che non sia stato richiesto l'annullamento dell'operazione, basterà verificare che *buttonIndex*, parametro passato dall'*UIAlertView*, corrispondente all'indice del pulsante premuto, non sia quello associato al tasto cancel; effettuiamo successivamente una semplice ricerca sul giorno della settimana inserito dall'utente nel primo campo di testo (indice = 0), come si può notare è stato utilizzato il casting su [*alertViewtextFieldAtIndex:0*].
Quando abbiamo identificato il giorno, potremo finalmente aggiungere la nuova azione nel corretto giorno della settimana con il relativo stato. Anche in questo caso avremmo potuto utilizzare l'operatore di doppia uguaglianza per verificare quale giorno era stato digitato. Operazione finale, e comunque obbligatoria, consiste nel notificare la *View*, la nostra tabella, che è avvenuta una modifica del *Model*, il nostro array.
Riceverete numerosi warning in fase di compilazione, questo è dovuto al fatto che stiamo usando metodi non resi pubblici dalla classe *UIAlertView*: ricordiamo che non è possibile rendere privato un metodo ma solo nasconderlo con alcuni artifici del linguaggio, quindi non c'è modo di impedirne effettivamente l'invocazione.

Andrea Leganza

L'AUTORE

Andrea Leganza
Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali, e non, su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certified Expert, e EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multimediali, anche con iPhone, con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it.

UNA SVEGLIA DIGITALE PER IPHONE

IN QUESTO ARTICOLO MOSTRIAMO COME REALIZZARE UNA SVEGLIA DIGITALE CHE CI AVVISERÀ DI IMPEGNI E SCADENZE IMMINENTI. SARÀ L'OCCASIONE DI APPROFONDIRE I CONCETTI LEGATI ALLA GESTIONE DELL'INTERFACCIA E DEL TIMER

CD WEB
sveglia.zip
cdrom.ioprogrammo.it

Fig. 1: Al termine del progetto avremo realizzato una sveglia digitale

REQUISITI

Conoscenze richieste
OOP

Software
MacOS X 10.5.4 o superiore, XCode

Impegno

Tempo di realizzazione

In questo articolo mostreremo come realizzare una sveglia digitale. Grazie a questo progetto avremo modo di introdurre alcune classi di estrema utilità: *NSTimer*, fornita dal framework Foundation, che insieme a *UIKit* rappresenta le fondamenta di tutte le applicazioni per iPhone, e *AVAudioPlayer*, presente nel framework *AVFoundation*, il cui scopo è consentire l'esecuzione di brani audio di "qualunque" dimensione. Poiché, come è risaputo, non è permesso nell'iPhone mantenere un'applicazione in background, si potrà utilizzare questa funzionalità di sveglia solo quando la nostra applicazione è effettivamente in esecuzione.

IL TIPO DI PROGETTO

Per realizzare questo software utilizzeremo un nuovo tipo di progetto fornito da *XCode* chiamato *Utility Application*: gli applicativi che vengono identificati con questo termine sono costituiti da due *UIViewController* contenenti una singola *UIView* ciascuno, che si alternano in base alla pressione di un preciso tasto. Il viewcontroller principale viene chiamato automaticamente *MainViewController*.
Il passaggio dall'una a l'altra schermata avviene con la pressione del pulsante *i* nel *MainView*, e con l'utilizzo di quello con label *done*, posizionato in alto a sinistra nella barra di navigazione di *FlipsideViewController*. Il sistema utilizzato per effettuare lo switch è relativamente complesso e non ne parleremo in questo articolo. Essendo la terza serie di articoli prenderemo per scontate tutte le pratiche necessarie per effettuare la creazione degli *IBOutlet* e per prelevare le informazioni da tali oggetti. L'intera business logic verrà implementata all'interno del *MainView Controller*, nel quale abbiamo accesso anche ai contenuti del *FlipsideViewController*.

IL COMPONENTE UIDATEPICKER

Inserendo un componente del tipo *UIDatePicker* all'interno della nostra interfaccia grafica, nel *FlipsideViewController*, forniremo all'utente la possibilità di selezionare una data, con una precisione a nostra discrezione; in questo caso impostiamo, tramite *Interface Builder*, il formato completo (giorno, mese, anno, ore, minuti) settando *mode* al valore "*Date and Time*", la lingua

italiana utilizzando il campo *locale*, e la precisione, con il campo *interval*, ad 1 minuto. *UIDatePicker* fornisce un metodo per ottenere la data selezionata, questa è un'istanza della classe *NSDate* e, quando torneremo al *MainView* provvederemo a memorizzarla in una cartella dell'iPhone, ed utilizzarla per verificare se l'allarme dovrà essere eseguito.

CONTROLLO NSTIMER

Un timer è un sistema che permette di cadenzare l'esecuzione di un metodo con precisi intervalli; ogni timer viene eseguito in un thread distinto. Attenzione, un *NSTimer* non garantisce che l'esecuzione del metodo associato avvenga sempre in un preciso istante, ma che avverrà in un istante pari o successivo a quello desiderato, questa limitazione è dovuta al fatto che, trovandoci in un ambiente multithread e multiprocess, dovremo condividere le risorse hardware con altri processi in esecuzione. In genere ciò avviene utilizzando uno dei cosiddetti "algoritmi di scheduling", i quali provvedono a fornire in maniera ciclica sufficiente tempo di calcolo sulla CPU a tutti i processi che ne facciano richiesta; potrebbe capitare quindi che, quando il nostro timer sia in procinto di scadere, avvenga un evento fuori dal nostro controllo che interrompa o rallenti l'esecuzione del nostro software per alcuni millisecondi, impedendogli di effettuare in tempo l'esecuzione del metodo. La precisione dichiarata è di circa 50/100 millisecondi, ma è comunque non garantita per le motivazioni appena citate.

```
NSTimer *timer = [NSTimer
                  scheduledTimerWithTimeInterval: 1
                                          target: self
                                        selector: @selector(handleTimer:)
                                        userInfo: nil
                                         repeats: YES];
- (void) handleTimer: (NSTimer *)
                  timer{ metodo invocato da NSTimer}
```

Con questo codice abbiamo realizzato un timer che viene avviato ogni secondo (o nei millesimi successivi), che allo scadere dell'intervallo esegue il metodo *handleTimer* e che si ripete all'infinito. Il metodo viene immediatamente avviato dopo la sua esecuzione e subisce un retain automatico. Quando, (e se), non avremo più bisogno di questa istanza basterà invocare su di essa il metodo *invalidate*, che provvederà ad effettuare su di essa un release automatico. Per il nostro scopo questo metodo dovrà verificare se è giunto il momento per eseguire l'audio dell'allarme, operazione effettuabile semplicemente confrontando la data attuale, ottenuta richiedendo il valore del campo *date* dalla classe *NSDate*, con quella prelevata dall'*UIDatePicker*:

```
- (void) handleTimer: (NSTimer *) timer {
 if ([alarmDate timeIntervalSinceNow]<=0)
 {
    //viene eseguito il suono dell'allarme
 }
}
```

timeIntervalSinceNow restituisce un valore positivo, i secondi mancanti alla data attuale, quando *alarmDate* è una data futura, mentre i valori sono negativi se *alarmDate* è ormai trascorso.
Abbiamo dovuto utilizzare sia l'uguaglianza che il simbolo di minore uguale perché, come è stato detto, non si può prevedere se il timer verrà eseguito nel preciso minuto in cui dovrebbe scattare l'evento, o in uno dei successivi.

LA MEMORIZZAZIONE DELL'ORA DELL'ALLARME

Dopo aver selezionato una data adoperando l'*UIDatePicker*, provvederemo a memorizzarla in una cartella locale all'iPhone e a caricarla ogni volta che il nostro applicativo verrà eseguito. All'interno del telefono esistono alcune cartelle liberamente accessibili e modificabili nelle quali potremo salvare qualunque tipo di informazione: *tmp* e *Documents*. La prima deve essere utilizzata per creare e gestire informazioni il cui tempo di vita è limitato alla singola esecuzione, mentre la seconda per tutti quei casi in cui un dato deve permanere per diversi avvii dell'applicativo. Esistono svariati metodi per ottenere la corretta posizione di queste cartelle, che sono uniche per ogni software, utilizzeremo il metodo consigliato da Apple quando si realizza un progetto che utilizza la tecnologia *Core Data* (della quale tratteremo in un prossimo articolo) :

```
- (NSString *)applicationDocumentsDirectory {
 return
 [NSSearchPathForDirectoriesInDomains(NSDocument
      Directory, NSUserDomainMask, YES) lastObject];
}
```

Questo codice restituisce il path, una stringa che rappresenta la posizione completa della cartella *Documents*. Questo valore potrà assumere valori diversi a seconda dell'applicativo, ma anche se vi troverete a utilizzare il tutto nel simulatore.
Ii questo caso punterà ad una sottocartella definita in: */Users/$NOMEUTENTE/Library/Application Support/iPhoneSimulator/User/Applications/*, oppu-

Fig. 2: Il MainView Controller che mosterà l'allarme

Fig. 3: Il FlipsideView Controller con l'UIDatePicker

re nel telefono. Per creare un path completo comprensivo del nome del file basterà comporre una stringa utilizzando la seguente procedura:

```
NSString *archivePath = [[self
                    applicationDocumentsDirectory]
  stringByAppendingPathComponent:@"sveglia.cfg"];
```

Con questa riga abbiamo realizzato con estrema semplicità un path per un file chiamato *sveglia.cfg* che verrà salvato e caricato quando richiesto. Per verificare se il file sveglia.cfg esiste già all'interno della cartella Documents, basterà semplicemente utilizzare il seguente metodo, *fileExistsAtPath*, fornito dalla classe *NSFileManager*, il quale restituirà *true* in caso affermativo, false in caso negativo:

```
if ([[NSFileManager defaultManager] fileExistsAtPath:
                                    archivePath])
{
 //il file esiste
}
else
{
 //il file non esiste
}
```

Queste informazioni verranno utilizzate per memorizzare la data in cui la sveglia dovrà essere avviata in modo da recuperare e utilizzare tale informazione ad ogni avvio.

NOTA

ULTERIORI APPROFONDIMENTI

Consultare la documentazione online denominata *Exception Programming Topics for Cocoa* per la gestione delle eccezioni e per l'elenco delle eccezioni di default.

AVAUDIOPLAYER

AVFramework, (*Audio Video Foundation Framework*), è un framework che consente di eseguire e anche registrare brani audio *AVAudioPlayer* è una delle classi fornite da tale libreria che consente l'esecuzione di un singolo file audio (uno per istanza); *AVAudioPlayer* consente di mettere in pausa e interrompere un audio, avere informazioni sulla sua durata e sulla posizione in cui è al momento l'esecuzione, consente infine di monitorare i vari livelli di volume assunti dall'audio. Questa classe accetta tutti i formati supportati dall'iPhone:

- AAC
- HE-AAC
- AMR (Adaptive Multi-Rate, a format for speech)
- ALAC (Apple Lossless)
- iLBC (internet Low Bitrate Codec, another format for speech)
- IMA4 (IMA/ADPCM)
- linear PCM (uncompressed)
- µ-law and a-law
- MP3 (MPEG-1 audio layer 3

Il formato suggerito dalla documentazione è 16-bit, little-endian, linear PCM di tipo CAF. È possibile convertire i proprio file audio in questo formato utilizzando il tool chiamato *afconvert* accessibile tramite la finestra di terminale di Mac OS:

```
/usr/bin/afconvert -f caff -d LEI16 {INPUT}{OUTPUT}
```

Nel caso di esecuzione multipla viene consigliato l'utilizzo del formato *IMA/ADPCM* (*IMA4*), mentre per l'ascolto di file singolarmente suggerisce *MP3*, *ALAC* (*Apple Lossless*), *AAC*, *IMA4*. Il primo file che viene eseguito accede direttamente alle risorse hardware, mentre i successivi saranno eseguiti via software. Apple raccomanda nella documentazione di utilizzare questa classe per eseguire qualunque tipo di effetto audio, a meno di avere necessità di gestire in modo distinto i canali stereo, di avere una sincronizzazione precisa, o quando si utilizzano file provenienti da flussi esterni, come avviene ad esempio per le web radio. Apple fornisce numerosi framework oltre ad AVFramework:

- **Media Player framework**: per eseguire brani musicali, audio book, podcasts;
- **Audio Toolbox framework**: per eseguire audio con precise necessità di sincronizzazione, o analisi o conversione, incluso maggiore controllo sulle fasi di registrazione;
- **Audio Unit framework**: per utilizzare plugin audio;
- **OpenAL framework**: viene consigliato come la migliore soluzione per eseguire e gestire musiche per i videogiochi, e utilizza OpenAL 1.1.

Tornando ora a *AVAudioPlayer*, nel nostro progetto utilizzeremo un loop audio, il tipico scandire del tempo di un orologio a tempo, e un suono che avviserà dell'allarme. Prima di effettuare qualunque operazione è necessario aggiungere *AVFoundation.framework* tra i framework che utilizzerà il progetto e importarlo all'interno di *MainViewController* (*#import <AVFoundation/AVFoundation.h>*). Il codice per eseguire un file audio è relativamente breve: prima provvediamo a identificare il path completo della risorsa che ci interessa, (un file mp3 in questo caso), poi creeremo un'istanza di *AVAudioPlayer*, imposteremo il volume, il numero di esecuzioni e lo avvieremo:

```
NSString *path = [[NSBundle mainBundle]
        pathForResource:@"clock" ofType:@"mp3"];
```

```
AVAudioPlayer player = [[AVAudioPlayer alloc]
 initWithContentsOfURL:[NSURL fileURLWithPath:path]
                                        error:nil];
                    player.volume = 0.4f;
                    [player prepareToPlay];
                    [player setNumberOfLoops:-1];
                    [player play];
```

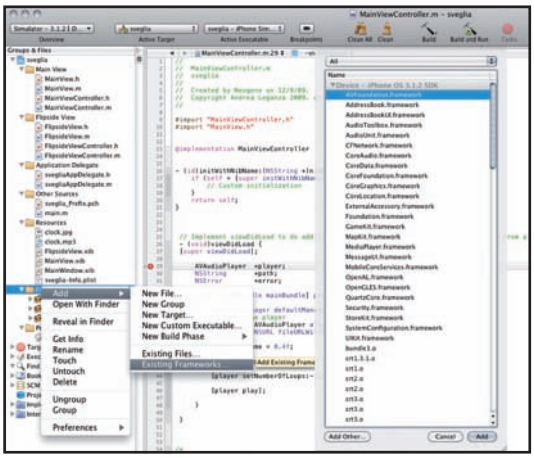

Fig. 1: La procedura necessaria per importare il framework

AVAudioPlayer ha un costruttore che accetta un URL, (che in questo caso sarà la posizione assunta nel nostro software dalla risorsa chiamata *clock.mp3*), e una variabile per memorizzare possibili errori (in questo progetto la abbiamo ignorata, impostandola a *nil*). Il campo *volume* è un *float* il cui intervallo è [*0,1*], dove con *0* si intende il silenzio, mentre con *1* il massimo valore consentito; il metoto *prepareToPlay* memorizza l'audio in un buffer prima di avviare l'esecuzione in modo da evitare interruzioni dovute al caching del file durante il play; *setNumberOfLoops*: può assumere un qualunque valore negativo per indicare un loop infinito, 0 per una singola esecuzione, *i* per (*i+1*) ripetizioni, inserendo 1 si avranno quindi due avvii successivi del suono. Se volessimo monitorare quando un suono è terminato basterà aggiungere il metodo *audioPlayerDid FinishPlaying*: all'interno della nostra classe, appartenente al protocollo *AVAudioPlayerDelegate*:

```
- (void)audioPlayerDidFinishPlaying:(AVAudioPlayer
                    *)player successfully:(BOOL)flag {
          //il suono è  terminato
}
```

GESTITE SITUAZIONI "ECCEZIONALI"

Chi utilizza linguaggi di alto livello, come JAVA o .NET, ha utilizzato generalmente in maniera esaustiva quei costrutti che consentono di catturare uno o più errori generati da una non corretta esecuzione del proprio applicativo.
Un'eccezione è il risultato di un comportamento anomalo, software o hardware, che il programmatore dovrebbe gestire per evitare che il proprio applicativo vada in crash e venga terminato. Per sapere se un metodo o una classe generano una o più eccezioni è necessario consultare la documentazione in linea. La sintassi necessaria per catturare queste eccezioni è estremamente semplice, basta inserire il codice che si vuole monitorare all'interno di un blocco di parentesi graffe a cui si antepone *@try*; con *@catch* si delimita quella parte di codice che dovrà gestire in modo opportuno l'arrivo dell'eccezione, ad esempio deallocando una risorsa, o cercando di risolvere il problema; *@finally* invece è un blocco che viene sempre eseguito e che generalmente viene adoperato per effettuare comuni operazioni di release e pulizia delle risorse utilizzate nel blocco *@try*. Poiché il codice racchiuso da *@finally* viene sempre eseguito risulta estremamente utile perché fornisce un unico punto in cui effettuare le tipiche operazioni di gestione delle risorse, invece di doverle ripetere al termine di *@try* e di *@catch*. Esistono numerosi tipi di eccezioni, tutte istanze di *NSException*, che si distinguono per il nome (il valore del campo *name*, di tipo *NSString*) che queste assumono. Oltre a quelle predefinite, ne esistono anche altre presenti in alcuni precisi contesti, e che sono comunque descritte approfonditamente nella documentazione; incapperete all'inizio molto spesso in *NSRangeException*, quando accederete a indici inesistenti di strutture dati (problema che non si presenta analizzandole utilizzando *Enumerators* e *Fast Enumerators* in ambienti *Thread Safe*), e *NSInvalidArgumentException*, quando passerete parametri non validi ad un metodo.

```
@try {
   //codice da monitorare
}
@catch (tipoeccezione *eccezione) {
    //nel caso avvenga un'eccezione di tipo
                              tipoeccezione viene gestita
}
@finally {
   //questo blocco di codice viene sempre eseguito
}
```

Nel caso di più eccezioni si potranno inserire diversi blocchi *@catch*, in questo caso viene prima analizzato se l'eccezione lanciata dentro *@try* è una versione realizzata ad hoc dal programmatore, di nome *CustomException* e in caso negativo viene confrontata con *NSException*:

NOTA

RIFERIMENTI WEB

Creazione dell'account, per scaricare l'SDK e consultare la documentazione: http://developer.apple.com/iphone/

```
@try {
}
@catch (CustomException *ce) {
}
@catch (NSException *ne) {
}
@finally {
}
```

Se si volesse gestire in un unico blocco *@catch* tutte le possibili eccezioni è sufficiente catturare quelle appartenenti alla classe *NSException* che, essendo la più generica da cui derivano tutte le altre, viene sempre riscontrata. Questa pratica è molto comune, ma spesso è più opportuno differenziare i tipi di eccezioni per avere un controllo più specifico di queste situazioni. Realizziamo un semplice blocco di codice in cui teniamo sotto controllo un metodo fornito dalla nostra classe che accetta come parametro un *NSMutableArray* che non ha alcun elemento (è infatti stato inizializzato con capacità nulla);

ECCEZIONE

Con il termine "eccezione" si intende un comportamento anomalo del proprio applicativo, generato da cause software o hardware; è sempre consigliato gestire le eccezioni per evitare un crash.

```
NSMutableArray *anArray = nil;

    array = [[NSMutableArray alloc]
                                    initWithCapacity:0];
    @try {
        [self metodo:anArray];
    }
@catch (NSException *exception) {
}
    @finally {
        [anArray release];
    }
```

Nel caso in cui il metodo che abbiamo invocato lanci un qualunque tipo di eccezione siamo in grado di catturarlo e gestirlo come più riteniamo opportuno. Poiché quando si presenta un'eccezione i vari blocchi *@catch* vengono analizzati in sequenza, si procede generalmente inserendo prima quelle eccezioni appartenenti a classi più specifiche, per poi arrivare alle più generiche, dove *NSException* rappresenta la più generica (è anche possibile utilizzare *id* come tipo di eccezione più generale ma non avrete probabilmente mai questa necessità): quando verrà rilevata un'eccezione nel blocco *@try* sarà cura dell'ambiente di esecuzione verificare se il primo *@catch* è adatto, oppure se dovrà procedere con il prossimo. In caso non ne venga trovato almeno uno, tale eccezione verrà inviata all'istanza che contiene quella dove è avvenuto tale evento e, in caso neppure questa sia in grado di gestirlo, continuerà il suo tragitto fino ad un certo punto, identificabile con il contenitore principale *UIApplicationMain* utilizzato all'interno di *main.m*, dopo di che si verificherà un crash del software. Quando viene segnalato nella documentazione in maniera esplicita che l'invocazione di un metodo può lanciare un'eccezione, è sempre consigliato provvedere a gestire tale evenienza. Nel caso non si desideri gestire un'eccezione è possibile rilanciarla, girarla all'oggetto che contiene l'istanza in cui ci troviamo, utilizzando *@throw*:

```
@try {
    //codice che lancia un'eccezione
}
@catch(NSException *e) {
    @throw; // rilancia l'eccezione
}
```

Anche in questo caso verrà utilizzata la procedura di ricerca progressiva di *@catch* in grado di gestirla.

L'utilizzo delle eccezioni dovrebbe essere ristretto solo a questo preciso scopo: gestire comportamenti anomali che richiedono un preciso intervento da parte dello sviluppatore, ma è anche possibile utilizzarle per rappresentare situazioni non anomale, come la pressione di alcune sequenze di tasti, che darebbero quindi inizio ad una precisa sequenza di eventi: tale comportamento, sebbene possibile, viene sconsigliato da Apple.

NSException è corredato di un campo chiamato *userInfo*, un *NSDictionary* nel quale è possibile inserire qualunque tipo di informazione allo scopo di passare alcuni dati al *@catch* che provvederà ad analizzarlo; tale utilizzo verrà spiegato in un altro articolo dove mostreremo come realizzare eccezioni customizzate. Per concludere, può risultare utile mostrare a schermo (in genere viene utilizzato un *UIAlertView*), o in un log, come nel prossimo esempio mostrato, la causa dell'errore e il tipo di eccezione, *NSException*, e conseguentemente tutte le classi derivate, fornisce due campi stringa proprio per questo scopo, *reason* e *name*:

```
@catch(NSException *e) {
    //gestione eccezione
    NSLog (@"EXCEPTION:%@
                                (%@)",e.reason,e.name);
}
```

LA CLASSE NSARCHIVER

La classe *NSArchiver* fornita da Cocoa è in grado di convertire in byte e da byte (in nume-

rosi linguaggi viene adoperato il termine *serializzare* e *deserializzare*), istanze di oggetti, scalari, strutture, stringhe e array, ma non consente di effettuare questa operazione per *union*, *void **, puntatori a funzione e sequenze di puntatori. Per quanto riguarda gli *NSArray* e gli *NSDictionary* questi devono contenere solo le strutture dati supportate per essere serializzate/deserializzare. La documentazione è sufficientemente esplicita a riguardo:

Only instances of NSArray, NSDictionary, NSString, NSDate, NSNumber, and NSData (and some of their subclasses) can be serialized. The contents of array and dictionary objects must also contain only objects of these few classes.

Per chi non fosse pratico con tali concetti, queste operazioni permettono di memorizzare lo stato di un'istanza su un determinato file. Adoperando poi la deserializzazione è possibile recuperare questo stato e riutilizzarlo per ripristinare l'istanza nella stessa configurazione precedente. Utilizziamo questa classe per memorizzare all'interno della memoria del telefono la data impostata dall'utente per la sveglia; perché questa procedura funzioni correttamente è necessario che la classe da cui deriva l'istanza su cui utilizziamo *NSArchiver* sia conforme al protocollo chiamato *NSCoding* il quale richiede che vengano implementati i seguenti metodi:

- *(void)encodeWithCoder:(NSCoder *)encoder (formale)*
- *(id)initWithCoder:(NSCoder *)decoder*

Solo *encodeWithCoder* viene esplicitamente richiesto, mentre *initWithCoder*, che provvede alla deserializzazione, è opzionale. Non mostreremo come realizzare il corpo di questi metodi perché la classe che dovremo memorizzare nel telefono, *NSDate*, è già conforme a questo protocollo. Per capire se una classe supporta *out of the box* questa tecnica basterà consultare la documentazione per verificare se *NSCoding* appare tra i protocolli supportati. Questa procedura può essere utilizzata per memorizzare informazioni arbitrarie in maniera estremamente semplice, in un videogioco o in un applicativo, esentando dall'obbligo di creare delle proprie strutture dati, come file xml o di testo, per organizzare tali dati. Per memorizzare la data che l'utente selezionerà adoperando l'*UIDatePicker* basterà utilizzare il metodo della classe *NSKEyedArchiver archive RootObject*: *toFile*: che consente di effettuare la serializzazione immediatamente. Questo metodo restituisce *true* in caso di salvataggio completato, *false* altrimenti.

```
NSString *archivePath = [[self
                    applicationDocumentsDirectory]
   stringByAppendingPathComponent:@"sveglia.cfg"];
NSString storeResult;
if ([NSKeyedArchiver archiveRootObject:alarmDate
                                toFile:archivePath])
{
        storeResult = @"Configurazione salvata con
                                successo.";
}
else
{
        storeResult = @"Impossibile salvare il file!";
}
```

Come si può vedere, salvare lo stato di un'istanza all'interno di un file locale è un'operazione estremamente semplice; riottenere invece la data all'avvio dell'applicativo comporta l'utilizzo di @try/@catch poichè, come viene esplicitato dalla documentazione: *This method raises an NSInvalidArgument Exception if the file at path does not contain a valid archive.* È possibile evitare di utilizzare questa pratica verificando prima l'esistenza del file utilizzando il metodo *fileExistsAtPath* fornito dalla classe *NSFile Manager*, mostrato in un precedente paragrafo.

```
@try
{
        if ((alarmDate = [NSKeyedUnarchiver
            unarchiveObjectWithFile:archivePath])){
        [alarmDate retain];
        ....
 }
@catch (NSException * e)
{
        //Ignoriamo l'eccezione di tipo
   NSInvalidArgumentException che viene lanciata nel
                caso sveglia.cfg non venga trovato
 }
```

CONCLUSIONI

In questo articolo abbiamo introdotto numerosi argomenti interessanti che consentono di realizzare soluzioni anche molto complesse, giochi compresi. Il progetto annesso alla rivista è completo e mostra come si integrano tutti questi concetti. Nei prossimi articoli continueremo questo viaggio nell'universo iPhone e Objective-C. Buona programmazione.

Andrea Leganza

L'AUTORE

Andrea Leganza
Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali, e non, su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certified Expert, e EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multimediali, anche con iPhone, con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it.

UN ACCELEROMETRO PER AMICO

L'ACCELEROMETRO PRESENTE NELL'IPHONE È UNO DEI COMPONENTI PIÙ EFFICACI NEL CONSENTIRE ALL'UTENTE UN'INTERAZIONE CON GIOCHI E APPLICATIVI MAGGIORMENTE SEMPLICE E INTUITIVA. VEDIAMO COME INTEGRARLO NELLE NOSTRE APPLICAZIONI

CD WEB
accelerometro.rar
cdrom.ioprogrammo.it

REQUISITI

Conoscenze richieste
OOP - Objectivive-C

Software
MacOS X 10.5.4 o superiore, XCode

Impegno

Tempo di realizzazione

In questo articolo andremo a realizzare un progetto minimale che ci consentirà comunque di approfondire un argomento molto interessante: ricevere, analizzare, e conseguentemente utilizzare i dati provenienti da uno dei sensori più utili presenti sia nell'iPhone che nell'iPod Touch: l'accelerometro. Questo piccolo componente si trova in un numero sempre crescente di dispositivi, merito anche del successo ottenuto da quando è stato inserito nel controller della Wii, il *wiimote*. Nell'iPhone questo componente viene principalmente adoperato per automatizzare la rotazione dell'interfaccia grafica di 90° o 180°. Con il rilascio della versione 3 del firmware (e del relativo SDK) è stata introdotta la funzionalità di *shake*, di scuotimento, che viene utilizzata per annullare l'operazione corrente, oppure per ripetere quella precedente, ma che si presta a centinaia di diverse interpretazioni, dipendenti solo dalla fantasia dello sviluppatore. Questo componente è uno di quelli che necessita di un dispositivo reale (non basta il simulatore software) per essere testato: simulare i suoi dati, utilizzando valori casuali o con una certa pianificazione, non sarà mai in grado di fornire un feedback accurato che consenta di constatare l'effettiva efficacia di come si interpretano tali dati. L'accelerometro è un componente elettronico realizzato dalla STMicroelectronics, ed è un modello a tre assi, simile nel funzionamento a quello installato nel controller della Wii, il Wiimote, ma in grado di rilevare teoricamente accelerazioni di maggiore intensità (fino a 8g); il sensore è in grado di funzionare in due modalità +/-2g e +/-8g, nel secondo caso si ha una perdita di precisione di circa quattro volte rispetto alle misurazioni ottenibili quando in modalità +/-2g, e per tale motivo Apple ha deciso di limitare il range di valori misurabili dal dispositivo per consentire di ottenere una maggiore accuratezza delle misurazioni. È quindi possibile ottenere solo valori compresi tra -2g e +2g, con una precisione di circa *0,018g*, e purtroppo non è permesso modificare attraverso l'API quale modalità il telefono deve utilizzare (2g o 8g).

INTERFACCIA GRAFICA: LA ROTAZIONE

Quando abbiamo trattato di *UIViewController* e *UITable* non abbiamo volutamente accennato a questa possibilità di ruotare l'interfaccia grafica in base all'effettiva posizione del telefono perché si è deciso di destinare tali informazioni al presente articolo. Queste rotazioni vengono gestite semplicemente consultando una variabile, senza interrogare direttamente l'accelerometro. La visualizzazione "landscape", orizzontale, è estremamente utile in numerosi contesti, non fornire ad un utente tale possibilità può avere un impatto negativo sull'opinione degli utilizzatori del vostro software e quindi, nel caso decidiate di vendere il vostro applicativo, potrebbe portare ad una riduzione delle vendite dovute a feedback non positivi. È estremamente semplice abilitare la propria interfaccia in modo che svolga tale operazione in maniera automatica, basterà decommentare un metodo, chiamato *shouldAutorotateToInterface Orientation*, creato automaticamente quando si realizza un *UIViewcontroller*, o una classe derivata, adoperando il wizard, e configurare a quali tipi di rotazioni tale controller dovrà reagire:

```
// Override to allow orientations other than the
                                default portrait orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:
                (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation ==
                                UIInterfaceOrientationPortrait);
}
```

Dopo aver decommentato il metodo analizziamo sia il parametro che riceve che il suo codice interno: *interfaceOrientation* è di tipo *UIInterface Orientation*, una *enum* in linguaggio C, i cui possibili valori sono i seguenti:

```
typedef enum {
    UIDeviceOrientationUnknown,
    UIDeviceOrientationPortrait,          // Device
```

```
        oriented vertically, home button on the bottom
    UIDeviceOrientationPortraitUpsideDown,  // Device
        oriented vertically, home button on the top
    UIDeviceOrientationLandscapeLeft,       // Device
        oriented horizontally, home button on the right
    UIDeviceOrientationLandscapeRight,      // Device
        oriented horizontally, home button on the left
    UIDeviceOrientationFaceUp,              // Device
        oriented flat, face up
    UIDeviceOrientationFaceDown             // Device
        oriented flat, face down
} UIDeviceOrientation;
```

Quando otteniamo un orientamento di tipo *portrait* significa che il telefono è tenuto in posizione verticale, sia quando il pulsante home del telefono punta verso il basso, che quando il telefono è capovolto e tale pulsante punta verso l'alto (*UpsideDown*); landscape ovviamente indica che il telefono è tenuto orizzontalmente, con il pulsante home posizionato sul lato sinistro (*LandscapeRight*), o sul lato destro

Fig. 3: La rotazione viene effettuata automaticamente

(*LandscapeLeft*); *FaceUp* e *Down* vengono utilizzati per notificare che il telefono ha lo schermo verso posizionato in basso, verso il pavimento, o verso l'alto. Queste modalità consentono di far reagire la propria interfaccia in base a 6 disposizioni predefinite del telefono, senza necessariamente analizzare i dati che provengono dal telefono e questa modalità di consultazione è la più indicata per modificare la grafica del proprio software. Il metodo viene interrogato dal sistema operativo del telefono che fornirà tale parametro e attenderà un valore booleano che gli consentirà di decidere se ruotare o meno l'interfaccia automaticamente. La scelta se ruotare o meno dipende da un semplice confronto tra il valore passato come parametro al metodo e i possibili stati del telefono. Decommentando questo metodo non si imposta ancora alcuna rotazione: per tale motivo, se volessimo far ruotare la nostra interfaccia grafica di 90 o 180 gradi, dovremo aggiungere, oltre a *UIInterfaceOrientationPortrait*, anche le altre modalità che vogliamo facciano scattare il meccanismo di rotazione. Ciò si effettua separando le modalità supportare utilizzato *OR* (simbolo ||):

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
        (UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    return (interfaceOrientation ==
                UIInterfaceOrientationPortrait ||
            UIInterfaceOrientationLandscapeLeft ||
            UIInterfaceOrientationLandscapeRight);
}
```

In questo modo abbiamo automatizzato la rotazione sia quando il software è verticale e quando è orizzontale in entrambi i versi.
Questo tipo di gestione dell'accelerometro è l'unica che è possibile testare parzialmente anche con il solo simulatore: basterà, tenendo premuto il tasto *mela*, operare sui i tasti cursore per far ruotare di 90° a sinistra e a destra l'interfaccia, e verificare conseguentemente come reagisce il software. Purtroppo le modalità *portraitupsidedown* e *faceup/down* non sono disponibili. La rotazione automatica si propaga anche ai componenti contenuti all'interno del *ViewController*, *UIView*, immagini e tutti gli altri inseriti dallo sviluppatore, ma non ha sempre l'esito sperato, ciò è dovuto al fatto che in alcune situazioni è necessario adoperare altri strumenti di modifica delle coordinate, che risultano più adatti (le cosiddette *Core Animation Transformations*, di cui parleremo in un altro articolo). Prendiamo come esempio un'immagine posizionata nel centro della nostra interfaccia.
Quando andremo a ruotare l'interfaccia otterremo un effetto simile a quello visibile in Fig.5.
La palla da rugby si è posizionata in maniera non corretta, un risultato alquanto indesiderato, vedremo in un prossimo articolo come effettuare delle trasformazioni sulle coordinate (le cosiddette trasformazioni affini) degli oggetti presenti nell'interfaccia che permetteranno di ottenere il risultato desiderato.

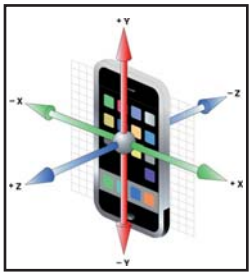

Fig. 1: La disposizione degli assi e dei versi positivi e negativi dell'accelerometro

Fig. 2: La comune visualizzazione verticale di una tabella

I MOTION EVENTS

L'SDK 3.0 ha introdotto il concetto di *motion events*, eventi generati dal movimento. Questi sono scatenati dopo alcune situazioni identificate dall'accelerometro, e non richiedono quindi misurazioni continue da parte dell'utente. Gli eventi disponibili sono in numero limitato:

- *motionBegan*
- *motionEnded*
- *motionCancelled*

Tutti i tre eventi ricevono due parametri: il primo, di tipo *UIEventSubtype*, consente di identificare se il movimento è stato identificato come appartenente ad una certa tipologia. Al momento l'unico

effettivamente utile è *UIEventSubtypeMotionShake* (*UIEventSubtypeNone* non fornisce alcuna informazione utile) il quale consente di identificare se l'utente ha agitato il telefono: se si volesse supportare la possibilità di annullare un'operazione di scrittura di una email ad esempio agitando il dispositivo, basterebbe verificare se la variabile motion assume valore pari a questo tipo di movimento con un semplice *if* e in caso affermativo agire di conseguenza (cancellando il testo, magari richiedendo con un *UIAlert* la conferma di tale operazione).
È possibile emulare lo shake tramite il simulatore adoperando la voce omonima presente nel menu hardware.

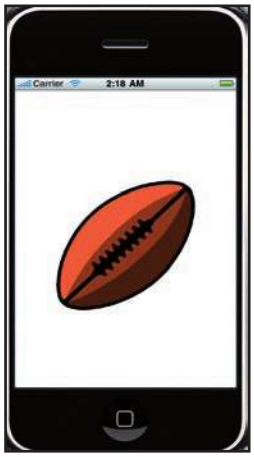

Fig. 4: Il pallone da rugby è al centro dell'interfaccia

```
- (void)motionBegan:(UIEventSubtype)motion
                                withEvent:(UIEvent *)event
{ //gestione evento}

- (void)motionEnded:(UIEventSubtype)motion
                                withEvent:(UIEvent *)event
{//gestione evento }

- (void)motionCancelled:(UIEventSubtype)motion
                                withEvent:(UIEvent *)event
{ //gestione evento

if (event.type == UIEventSubtypeMotionShake) {
    //se il movimento è di shake agiamo di conseguenza
  }
}
```

Il parametro *event* non ha alcuna utilità in questo contesto, viene utilizzato per altri tipi di eventi, quelli *touch* (il tocco dello schermo) e fornisce informazioni sulle coordinate dei dei vari punti in cui è avvenuta l'interazione dell'utente. L'utilizzo di *motionBegan* e *motionEnded* è immediato, mentre l'utilità di *motionCancelled* è meno intuitiva: questo metodo assume importanza quando un evento identificato come uno di tipo shake dura troppo tempo (oltre un secondo) oppure quando il framework *Cocoa Touch* richiede l'interruzione dell'analisi del movimento, in entrambi i casi questo blocco di codice dovrebbe contenere il codice per ripristinare lo stato del software prima delle modifiche effettuate all'interno di *motionBegan*.
Nel caso in cui il movimento non venga gestito in nessun *UIViewController*, e la proprietà *applicationSupportsShakeToEdit* del nostro applicativo sia impostata a *YES* (accessibile tramite il singleton *UIApplication* di cui parleremo in un prossimo articolo) verrà presentato un menu di scelta tra *undo* (annullamento) o *redo* (ripetizione). Questi tre metodi sono meno utili rispetto alla consultazione dei dati forniti dall'accelerometro ma nel caso dell'identificazione del movimento di shake forniscono il modo più semplice e veloce per supportare questo tipo di interazione.

Fig. 5: La rotazione automatica si è rivelata scarsamente efficace, generando un risultato indesiderato nella maggior parte dei casi

NOTA

SPECIFICHE HARDWARE

A questi indirizzi è possibile consultare la documentazione dei due modelli di sensori utilizzati nelle diverse versioni dell'iPhone e dell'iPod Touch:
LIS302DL:
http://www.st.com/stonline/products/literature/ds/12726/lis302dl.pdf
LIS331DL:
http://www.st.com/stonline/products/literature/ds/13951.pdf

IL PRELIEVO DELLE INFORMAZIONI

Ora che abbiamo mostrato due situazioni in cui fare uso in maniera implicita delle informazioni provenienti dall'accelerometro, realizziamo il codice necessario per abilitare e leggere i suoi dati in modo esplicito. Ogni applicativo può accedere a una sola istanza della variabile che fornisce i dati dell'accelerometro, un singleton secondo la terminologia object oriented, tale istanza appartiene al tipo di dato *UIAccelerometer*; per abilitare la ricezione dei dati si provvede a impostare se stessi, la classe in cui si analizzano i dati, come delegate dell'accelerometro, mentre per disabilitare la ricezione è sufficiente reimpostare a *nil* tale delegate.

```
int frequency = 50;
UIAccelerometer*  myAccelerometer;

-(IBAction)configurazioneAccelerometro
{
    myAccelerometer = [UIAccelerometer
                                sharedAccelerometer];
    myAccelerometer.updateInterval = 1 / frequency;
    //Impostiamo il delegate
    myAccelerometer.delegate = self;
   //Da questo momento verranno inviati i dati
        dall'accelerometro tramite l'invocazione del metodo
                                accelerometer: didAccelerate
 }
```

Impostando la classe in cui si trova come delegate si dichiara di rispettare il protocollo chiamato *UIAccelerometerDelegate*, come suggerito in articoli precedenti è sempre consigliato dichiarare tale operazione nel file di interfaccia *.h*:

```
@interface accelerometroViewController :
                UIViewController <UIAccelerometerDelegate>
```

La variabile *myAccelerometer* viene creata richie-

dendo al singleton *UIAccelerometer* la sua istanza, tramite il metodo *sharedAcelerometer*. È possibile poi prelevare dati a diverse frequenze, in questo caso abbiamo scelto *50Hertz*, ciò significa il prelievo di una misurazione del sensore ogni 20 millisecondi. Per chi non avesse conoscenza di queste definizioni 1Herz indica una misurazione ogni secondo, 2 Herz 2 al secondo, 10 Hertz 10 al secondo e così via.
Il range di valori consentiti e consigliati da Apple sono:

- 10-20 Herz: per misurare il verso del telefono
- 30-60 Herz: per giochi e applicativi in cui è richiesta maggiore precisione
- 70-100 Herz: per misurazione con variazioni di posizione molto elevata

I valori verso i 100 Herz non vengono quasi mai utilizzati perché raramente è necessaria una tale precisione. L'asse z assumerà un valore prossimo a +1 quando il telefono sarà appoggiato su una superficie piana, parallela al pavimento, -1 quando lo schermo toccherà il piano; *y* assumerà valore -1 quando il telefono sarà in posizione portrait con il pulsante home in basso, +1 quando il pulsante home sarà rivolto verso l'alto; *x* varrà -1 quando lo schermo sarà rivolto verso sinistra e +1 verso destra. Tutti i valori compresi tra + e -1 per tutti gli assi verranno assunti quando non si posiziona il telefono perfettamente perpendicolare al piano. La somma delle tre misurazioni deve valere in valore assoluto circa 1. Dopo aver configurato la modalità di ricezione dei dati, notifichiamo alla variabile che siamo in grado di ricevere e gestire le sue informazioni, per tale motivo la informiamo che siamo un delegate; a questo punto è necessario implementare il metodo che l'accelerometro invocherà ad ogni aggiornamento:

```
- (void)accelerometer:(UIAccelerometer
        *)accelerometer didAccelerate:(UIAcceleration
                                    *)acceleration

{
   ...
```

Avevamo detto che i dati provenienti dall'accelerometro erano di tipo *double*, ma in questo caso abbiamo adoperato *UIAccelerationValue*, ebbene questo tipo di dato è intercambiabile con *double*, essendo un suo alias:

```
typedef double UIAccelerationValue;
```

Per finire, creiamo un metodo che verrà invocato quando si desidera che termini il rilevamento:

```
-(void) terminaAnalisi {
        myAccelerometer.delegate = nil; }
```

Nel caso si desideri gestire all'interno di un solo *UIViewController* queste misurazioni è obbligatorio fare in modo che il delegate sia impostato a *nil* prima che il viewcontroller venga rimosso dallo stack di controller. Se ci troviamo in un sistema di navigazione tra controller, poichè altrimenti si rischia il verificarsi di crash casuali dovuti all'invocazione del metodo *accelerometer: didAccelerate* su un'istanza non più esistente (con il verificarsi della chiamata ad uno zombie quindi). Per visualizzare immediatamente i dati provenienti dalla misurazioni dei tre assi basterà creare tre *UILabel*, impostarle come *IBOutlet*, e modificare il testo contenuto in base alle rilevazioni:

```
- (void)accelerometer:(UIAccelerometer
        *)accelerometer didAccelerate:(UIAcceleration
                                    *)acceleration
{
   UIAccelerationValue x, y, z;
   x = acceleration.x;
   y = acceleration.y;
   z = acceleration.z;

//Aggiornamento del testo dei tre UILabel
        xLabel.text = [NSString
                        stringWithFormat:@"%g",x];
        yLabel.text = [NSString
                        stringWithFormat:@"%g",y];
        zLabel.text = [NSString
                        stringWithFormat:@"%g",z];
}
```

UNA WII COME TESTER

Anche se all'inizio dell'articolo abbiamo dichiarato che non è possibile simulare senza il dispositivo reale i dati dell'accelerometro, in realtà esistono alcune soluzioni che consentono di bypassare questa richiesta, una di queste consiste nell'adoperare un controller della Wii, collegarlo via Bluetooth e fornire tali informazioni attraverso un web server, che verrà consultato da codice appositamente realizzato nel proprio applicativo sull'iPhone. Questa soluzione comporta sia la realizzazione del codice client nel telefono che di quello server al quale fa riferimento il dispositivo e non è quindi un'operazione che un utente non pratico con un altro linguaggio di programmazione (ad esempio C#) potrebbe realizzare in maniera autonoma in tempi relativamente brevi. Inoltre si deve tener conto che il sensore del wiimote invia dati fino a 3g, mentre, come abbiamo precisato in precedenza, nell'iPhone si arriva fino a 2g.

Andrea Leganza

NOTA

RIFERIMENTI WEB
Creazione dell'account, per scaricare l'SDK e consultare la documentazione: http://developer.apple.com/iphone/

L'AUTORE

Andrea Leganza
Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali, e non, su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certified Expert, e EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multimediali, anche con iPhone, con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it.

IPHONE: GESTIRE IL MULTI-TOUCH

LO SCHERMO MULTITOUCH CONSENTE DI INTERAGIRE CON UN DISPOSITIVO SENZA LA NECESSITÀ DI FORNIRE INGOMBRANTI TASTIERE. IMPARIAMO A INTERCETTARE LE AZIONI DEGLI UTENTI E A GESTIRLE IN MODO DA NON FAR RIMPIANGERE TASTIERA E MOUSE

L'evoluzione tecnologica e la conseguente discesa dei prezzi hanno consentito ai produttori di dispositivi mobili di inserire nei propri prodotti uno o più schermi touch screen: quello che prima era un componente riservato a contesti specializzati, è divenuto sempre più frequente nell'elettronica di largo consumo. Nell'interfaccia dell'iPhone il multitouch consente di interagire in molti modi con i componenti grafici:

- tocco singolo (single tap)
- tocchi multipli (multi tap)
- trascinamento (drag)
- scrolling verticale (swipe verticale)
- strisciata (swipe orizzontale)

Il termine *Swipe*, utilizzato nella documentazione iPhone, è sinonimo del più comune "scrolling", ed è quindi quell'operazione adoperata generalmente per passare da un contenuto all'altro, immagini o video ad esempio, e che consiste in un repentino movimento di uno o più dita da destra a sinistra o viceversa; In questo articolo vedremo di accedere ai dati che questo dispositivo è in grado di fornire allo sviluppatore, e li utilizzeremo per i nostri scopi.

COME FUNZIONA UN TOUCH SCREEN

Il multi-touch presente nell'iPhone è un insieme di strati costituiti da diversi materiali e in alcuni livelli è percorso da componenti elettrici che, quando vengono a contatto con gli strati superiori, identificano tale situazione come la pressione da parte dell'utente di una precisa area dello schermo. Gli strati necessari per il funzionamento del multitouch sono interposti tra una copertura antiriflesso, quella che effettivamente l'utente tocca con le proprie dita, e il display LCD vero e proprio. Cosa succede quando un utente tocca lo schermo? I vari strati superficiali collidono nei punti in cui avvengono le pressioni, questi eventi generano dei segnali elettrici abbastanza imprecisi a causa di normali interferenze elettriche, che rappresentano aree di schermo di circa 1 cm di lato (tale è la dimensione dell'area che andiamo a toccare effettivamente), su queste informazioni vengono effettuate delle stime per identificare al meglio la regione dove effettivamente l'utente voleva interagire, infine vengono calcolate le coordinate del centro di queste aree e vengono inviate, se richiesto, al vostro software, altrimenti vengono ignorate.

UIEVENT E UITOUCH

Ogni componente può rispondere alle interazioni dell'utente, purché discenda dalla classe *UIResponder*, *UIViewController*, *UIView*, e conseguentemente le loro classi figlie (in pratica tutti i componenti grafici), derivano da tale padre, e per tale motivo sono pienamente autorizzate a gestire tali eventi. Ovviamente, poiché un *UIViewController* contiene al suo interno un *UIView*, potrà intercettare il tocco prima delle view e gestirlo senza demandare tale operazione ad una delle sue view.

I seguenti quattro metodi:

```
- (void)touchesBegan:(NSSet *)touches
                                withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches
                                withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches
                                withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet *)touches
                                withEvent:(UIEvent *)event
```

sono quelli, forniti dalla classe *UIResponder*, che ci consentono di intercettare i tocchi effettuati dall'utente: non è necessario alcuna operazione aggiuntiva, (come impostare protocolli o delegate per iniziare a gestirli). L'ultimo metodo è meno intuitivo, viene invocato dal sistema quando si scatenano eventi fuori dal controllo dell'utente, come una

CD WEB
iphone149.zip
cdrom.ioprogrammo.it

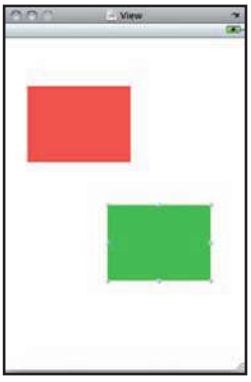

Fig. 1: Una semplice interfaccia grafica per realizzare un sistema di drag and drop

REQUISITI

Conoscenze richieste
OOP - Objectivive-C

Software
MacOS X 10.5.4 o superiore, XCode

Impegno

Tempo di realizzazione

possibile chiamata, una rimozione della view su cui si sta analizzando il tocco (ad esempio per un retain che ne invoca il distruttore), o altra notifica come quello di batteria in esaurimento, che annullano il completamento delle operazioni di tocco iniziate dall'utente, in questo metodo generalmente si ripristina il sistema allo stato interno precedente, corrispondente a quello trovato quando è stato invocato per la prima volta il metodo *touchesBegan*. Il primo parametro di questi metodi, *NSSet*, è un'istanza di una classe il cui scopo è quello di raccogliere in sè più oggetti in maniera non ordinata. In questo caso al suo interno troviamo tutte istanze della classe *UITouch*, ognuna a rappresentare le diverse dita che toccano lo schermo.
La classe *UITouch* fornisce informazioni riguardo

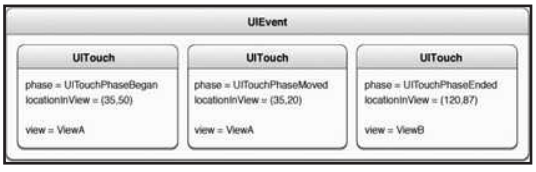

Fig. 2: Il contenuto di una istanza di tipo UIEvent

la posizione attuale del tocco (un punto di tipo *CGPoint* e coordinate (*x,y*) ottenuto con la chiamata al metodo (*locationInView:*), quella precedente (*previousLocationInView:*), il numero di pressioni ripetute (proprietà *tap*) del dito sullo schermo ed effettuate in breve sequenza (proprietà *tapCotunt*), la view in cui questo è stato identificato (proprietà *view*), l'istante temporale in cui tali coordinate sono cambiate (proprietà *timestamp*, utile per misurare la durata di un movimento, ad esempio), e la fase del tocco (proprietà *phase*), informazione che consente di identificare lo stato del tocco:

UITouchPhaseBegan,
UITouchPhaseMoved,
UITouchPhaseStationary,
UITouchPhaseEnded,
UITouchPhaseCancelled,

Quando ci troviamo in un progetto in cui non ci interessa se uno o più dita toccano lo schermo, ma vogliamo sapere la posizione di uno qualsiasi dei tocchi, basterà invocare il seguente metodo, reso disponibile dalla classe *NSSet*, all'interno di uno dei metodi *touches**:

```
UITouch *touch = [touches anyObject];
```

il quale restituisce un'istanza in modo pseudo-casuale (non è detto che sia sempre diversa per ogni invocazione successiva); ovviamente se esiste un solo tocco *anyObject* restituirà sempre questo. Nell'articolo precedente avevamo parlato di come gestire i dati che provenivano dal sensore, l'accelerometro, presente nel dispositivo, e avevamo trattato della famiglia di eventi definiti con la struct *UIEventType* che raccoglie al suo interno sia gli input di movimento (misurati dall'accelerometro) che quelli di tocco:

```
typedef enum {
    UIEventTypeTouches,
    UIEventTypeMotion,
} UIEventType;
```

Il secondo parametro di tutti i metodi *touches** è di tipo *UIEvent* e assumerà come tipo ovviamente il valore *UIEventTypeTouches*. L'istanza appartenente alla classe *UIEvent* rappresenta quindi lo stato attuale delle dita che sono a contatto con lo schermo, e consente di identificare univocamente le singole istanze di *UITouch* e come è variato il loro stato. Per ottenere i tocchi relativi ad una determinata view basta invocare all'interno di uno dei metodi *touches** su tale *UIEvent* il metodo *touchesForView:*.
Come riesce il dispositivo a identificare in quale *UIView* viene effettivamente effettuato il tocco con una certa coordinata? Questa ricerca avviene semplicemente effettuata in maniera iterativa: si parte dal contenitore principale, che racchiude in se tutti gli oggetti visuali del nostro applicativo, la *UIWindow*, e si procede "scendendo" tra gli *UIViewController* e le rispettive *UIView* effettuando su di queste il metodo *pointInside:withEvent*. Tale procedura prosegue per ogni *UIView* che restituisce *YES* a tale chiamata, fino a raggiungere l'ultima *UIView* che diventerà la responsabile della gestione e dell'interpretazione del tocco: tale sequenza di *view* e *viewcontrollers* prende il nome di "responder chain", catena di risposta. Nel caso l'ultima *view* non fosse configurata per gestire il tocco il

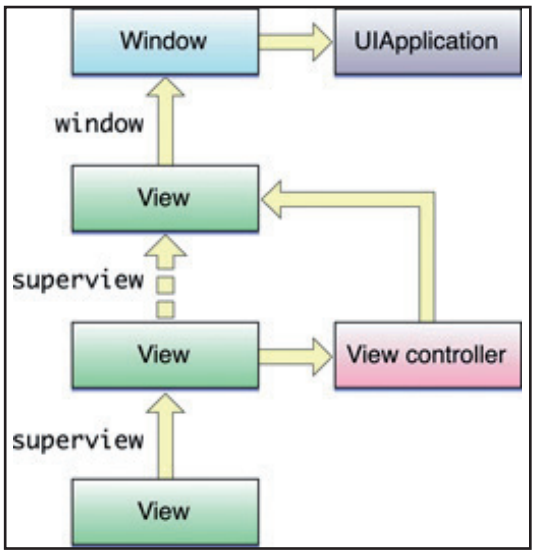

Fig. 3: La responder chain dell'iPhone

NOTA

TOUCH -SCREEN

Per una descrizione più approfondita del funzionamento del touch-screen dell'iPhone: http://electronics.howstuffworks.com/iphone1.htm

NOTA

RIFERIMENTI WEB

Creazione dell'account, per scaricare l'SDK e consultare la documentazione: http://developer.apple.com/iphone/

metodo di ricerca risalirà tra le *UIView* (e i rispettvi controllers) che la contengono in cerca di una in grado di "reagire" a tale operazione; nel caso peggiore, quello in cui si ritrovasse di nuovo nell'*UIWindow*, tale evento verrà ignorato.

ABILITARE E DISABILITARE L'INTERAZIONE

Nel caso in cui non si desideri che una *view* riceva uno o più eventi dovuti al tocco da parte dell'utente è possibile disabilitare tale interazione in diversi modi attraverso l'utilizzo di proprietà o metodi forniti dalla classe *UIView*:

- Configurandola come trasparente (ex: *myUIView.alpha=0.0*);
- Configurandola come nascosta (ex: *myUIView.hidden=YES*);
- Impostando la sua proprietà *userInteraction Enabled* a NO: (ex: *myUIView.userInteraction Enabled=NO*);

Nel caso si decidesse di disabilitare completamente l'interazione per l'intero applicativo, ad esempio quando si effettuano delle animazioni sarebbe sufficiente invocare prima di tale situazione *beginIgnoringInteractionEvents* e successivamente *endIgnoringInteractionEvents* sul singleton *UIApplication* (è un'istanza di una classe unica durante tutta la vita dell'applicativo) nel seguente modo:

```
[[UIApplication sharedApplication]
                        beginIgnoringInteractionEvents];
//Eventi che non richiedono interazione con l'utente
[[UIApplication sharedApplication]
                        endIgnoringInteractionEvents];
```

Per default i tocchi multipli non vengono gestiti automaticamente, sono perciò ignorati e vengono passate all'applicativo solo quelle informazioni relative al primo tocco rilevato; per abilitare il multitouch è necessario invocare nel proprio codice, quando necessario, il metodo *multipleTouchEnabled* sulla UIView che dovrà gestirli. Se si desidera fare in modo che solo una UIView risponda a tutti gli eventi si dovrà impostare la sua proprietà a *exclusiveTouch* a YES.

LE SEQUENZE DI TAP

Multitouch non significa solo gestire i tocchi delle dita contemporaneamente senza identificare la sequenza con cui questi sono avvenuti, nell'iPhone OS è presente una logica in grado di monitorare e analizzare diversi tipi di interazioni, sia in base al numero che alla direzione del movimento effettuato dalle dita che in quel preciso momento sono a contatto con lo schermo.
La sequenza tipo di un evento di tipo singolo tocco è costituito dai seguenti passi:

1- un dito tocca lo schermo;
2- il dito viene mosso mantenendolo a contatto con lo schermo (opzionale)
3- il dito viene rimosso

Se il movimento, (il secondo passo descritto nella lista), è un'operazione opzionale, la prima e la seconda avvengono obbligatoriamente in un intervallo di tempo che può essere più o meno breve, nel caso in cui si effettui la tipica operazione di tap, termine che sta ad indicare la pressione singola o più volte delle dita, siamo in grado in maniera semplice di ottenere il numero di questi tocchi senza necessariamente dover utilizzare una variabile per contabilizzarli.
La sequenza tipo di un evento multi tocco è costituito dai seguenti passi:
1 - un primo dito tocca lo schermo;
2- un secondo dito tocca lo schermo (il numero delle dita può essere superiore a due);
3 - uno o entrambi effettuano un movimento;
4 - in successione vengono rimossi entrambi

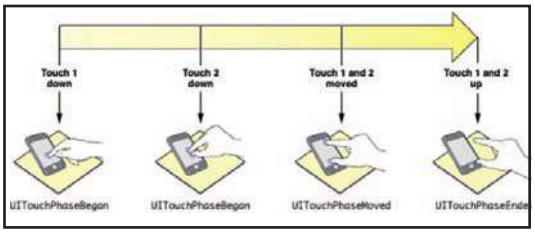

Fig. 5: Una tipica situazione di sequenza di eventi

SINGLE-TAP E MULTI-TAP

Come è stato detto precedentemente è possibile conoscere con estrema facilità il numero di tocchi effettuati in un breve intervallo di tempo dall'utente, basta consultare la proprietà *tapCount*, in genere ciò viene fatto all'interno del metodo *touchesEnded* (quando uno o più dita vengono rimosse quindi).

```
- (void)touchesEnded:(NSSet *)touches
                        withEvent:(UIEvent *)event {
    for (UITouch *touch in touches) {
        if (touch.tapCount == 1) {
```

Fig. 4: I due checkbox posizionati in basso consentono di cambiare il comportamento predefinito della view attraverso Interface Builder

```
    //do something  }
    else if (touch.tapCount == 2) {
          //do something     }
    else if (touch.tapCount == 3) {
//do something}
//etc tapCount>3
    }
}
```

IDENTIFICHIAMO E GESTIAMO LO "SWIPE"

Abbiamo detto che con il termine *swipe* si identifica lo scorrimento orizzontale/verticale di uno, o più dita. Per supportarlo all'interno delle proprie *UIView* (o *UIViewController*) è necessario effettuare alcuni calcoli che identifichino se lo spostamento delle dita è dovuto ad un movimento volontario oppure ad un semplice tremolio del dispositivo in mano all'utente; per fare ciò si deve misurare in un intervallo di tempo come variano le coordinate del tocco e, nel caso tale delta sia superiore o uguale ad un valore predefinito, si può ritenere con una relativa certezza che l'utente stia effettivamente effettuando tale operazione. Vista l'inaccuratezza del movimento delle dita in una delle due direzioni, è opportuno considerare come *swipe* qualunque movimento che comporti uno spostamento all'interno di un ipotetico rettangolo , il cui centro geometrico è dato dalla coordinata del primo tocco. Studiamo il caso di *swipe orizzontale*.

Se il movimento verticale (sia verso l'alto che il basso) supererà una costante predefinita, *MASSIMODELTAVERTICALE*, il cui valore è espresso in pixel, non identifichiamo tale movimento come uno swipe. In tale situazione è probabile che l'utente voglia effettuare o uno scrolling in direzione obliqua oppure in orizzontale, ma per qualche motivo non è riuscito ad effettuarlo in maniera corretta. Di conseguenze se il movimento orizzontale non sarà maggiore di un certo numero di pixels, espresso dalla variabile *MINIMODELTAORIZZONTALE*, allora è probabile che il movimento identificato non sia volontario e lo ignoreremo. Variando tali estremi potremo decidere in che modo reagire ai diversi movimenti , forzando l'utente ad effettuare gesti molto precisi oppure consentendogli una maggiore libertà.

Fig. 6: Una tipica situazione di sequenza multipla di eventi

```
#define MINIMODELTAORIZZONTALE  8 //minimo
                          spostamento orizzontale in pixel
#define MASSIMODELTAVERTICALE    6 //massimo
                          spostamento verticale in pixel
```

Iniziamo memorizzando la posizione del primo tocco nella variabile chiamata *startTouchPosition*: è di tipo *CGPoint*, una semplice struttura *C*

```
struct CGPoint {
   CGFloat x;
   CGFloat y;
};
typedef struct CGPoint CGPoint;
```

I valori che *CGPoint* potrà rappresentare quando toccheremo lo schermo con un dito potranno appartenere a qualunque punto di coordinate (x,y), dove *x* è compreso tra 0 e 320 e y tra 0 e 480 (0,0 bordo alto a sinistra del telefono, 320,480 basso a destra), dovute alla risoluzione dell'LCD dell'iPhone (*CGPoint* può assumere comunque valori superiori o inferiori a questi intervalli ma ovviamente rappresentano posizioni esterne all'LCD e non visibili all'utente).

Per determinare se il movimento è contenuto all'interno dell'area utile, effettuiamo semplicemente il calcolo della differenza tra la coordinata precedente, misurata all'interno del metodo *touchesBegan*, e quella rilevata successivamente all'interno di *touchedMoved*: in valore assoluto, adoperando la funzione C *fabsf*:

```
CGPoint startTouchPosition;
- (void)touchesBegan:(NSSet *)touches
                          withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    startTouchPosition = [touch locationInView:self];
}
- (void)touchesMoved:(NSSet *)touches
                          withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    CGPoint currentTouchPosition = [touch
                          locationInView:self];
 if (fabsf(startTouchPosition.x -
                          currentTouchPosition.x) >=
                          MINIMODELTAORIZZONTALE &&
    fabsf(startTouchPosition.y - currentTouchPosition.y)
                          <= MASSIMODELTAVERTICALE) {
       if (startTouchPosition.x <
                          currentTouchPosition.x) {
          //swipe a destra
        }
       else {//swipe a sinistra}
}}
- (void)touchesEnded:(NSSet *)touches
                          withEvent:(UIEvent *)event {
   //azzeriamo la posizione iniziale
    startTouchPosition = 0.0;}
- (void)touchesCancelled:(NSSet *)touches
                          withEvent:(UIEvent *)event {
  //azzeriamo la posizione iniziale se è avvenuto un
                          evento che ha annullato l'operazione
    startTouchPosition = 0.0;}
```

Quando verrà alzato il dito, sarà sufficiente azzera-

re la posizione iniziale per essere di nuovo pronti a studiare i tocchi successivi. È possibile quindi, variando i due valori di delta predefiniti, creare aree verticali per supportare lo scrolling verticale, come avviene ad esempio per le *UITableView* o le *UIScrollView*.

GESTIRE IL TRASCINAMENTO

L'operazione di drag&drop, si rivela meno complicata rispetto a quella di swipe: basterà infatti identificare quale *UIView* è stata toccata e muoverla di conseguenza, in risposta agli spostamenti del dito. Esistono due metodi principale per gestire tale operazione:

gestire tale comportamento all'interno del *UIViewController*, che provvederà a spostare la *UIView*
gestire tale comportamento all'interno della *UIView* stessa, creando una classe figlia di *UIView*.

Nel nostro progetto abbiamo deciso di gestire l'interazione all'interno del viewcontroller: il pregio di questo approccio è dovuto alla possibilità di identificare possibili sovrapposizioni tra le view e agire di conseguenza (ad esempio realizzando un semplice algoritmo di analisi delle collisioni come viene fatto per i giochi) e quindi dovremo inserire i vari metodi *touches** nel suo file di implementazione (*.m*). Creiamo quindi il nostro progetto, di tipo *View-Based Application*. Creiamo due *IBOutlet* (ricordate di chiamare su entrambi il metodo *release* nel metodo *dealloc* allo scopo di evitare leaks), di nome *red* e *green* in questo esempio, all'interno del file di interfaccia (*.h*) della classe chiamata *progettoViewController*:

```
@interface progettoViewController : UIViewController
{
    IBOutlet UIView *red;
    IBOutlet UIView *green;
}
@end
```

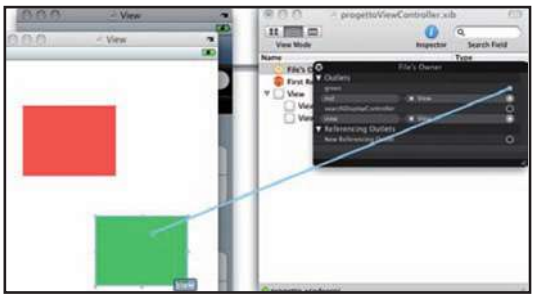

Fig. 8: L'operazione di associazione tra IBOutlet e UIView

Apriamo con interface builder il file chiamato *nomeprogettoviewcontroller.xib* e trasciniamo all'interno della view principale due ulteriori *UIView*, ridimensioniamole a nostro piacimento, e impostiamone i colori in modo da poterle distinguere visivamente. Premiamo il tasto destro del mouse sulla voce *Files Owner* e, tramite la solita operazione di trascinamento dei due IBOutlet, colleghiamoli con le due *UIView* appena create.
Ora non ci resta che implementare i soliti metodi per gestire l'operazione di trascinamento; per semplicità realizziamo solo il codice relativo all'evento di movimento che invoca di conseguenza il metodo *touchesMoved*: *withEvent*, gli altri in questo caso non sono necessari.

```
- (void)touchesMoved:(NSSet *)touches
                                withEvent:(UIEvent *)event
{
    UITouch *theTouch = [touches anyObject];

    CGPoint newPosition = [theTouch
                                locationInView:self.view];

    if ([self.view hitTest:newPosition withEvent:event]
                                        == red){
        red.center = newPosition;
    }
    else if ([self.view hitTest:newPosition
                                withEvent:event] == green){
        green.center = newPosition;
    }

}
```

Il funzionamento è abbastanza intuitivo: prima otteniamo la posizione del tocco, rappresentata dal solito *CGPoint*, successivamente dobbiamo decidere se questo punto è occupato da uno dei nostri due *UIView*. Per effettuare tale operazione interroghiamo la view principale, *self.view*, quella che contiene le due *UIView* figlie, invocando su di questa il metodo *hitTest*; *hitTest* è un metodo che restituisce come risultato quale *UIView* si trova in un determinato punto: basterà quindi verificare che questa corrisponda alla view chiamata *red* o *green*, per decidere su quale stia avvenendo il tocco. Per spostare la view selezionata basterà cambiarne il centro, utilizzando la proprietà *center* (sempre di tipo *CGPoint*), attribuendole valore pari alla posizione del tocco. Una soluzione alternativa al codice proposto per venire a conoscenza di quale view è al momento toccata consiste nell'utilizzare il metodo *touchesForView*: fornito da *UIEvent*. Abbiamo così mostrato i vari modi di interpretare l'interazione dell'utente in base al numero ed al tipo di touch.

Andrea Leganza

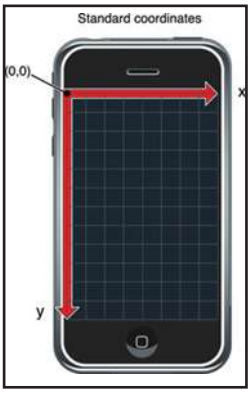

Fig. 7: Il sistema di coordinate dell'iPhone

L'AUTORE

Andrea Leganza
Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali, e non, su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certified Expert, e EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multimediali, anche con iPhone, con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it.

UN RSS READER SU IPHONE

DISTRIBUIRE INFORMAZIONI ATTRAVERSO LE NOSTRE APPLICAZIONI PUÒ ESSERE MOLTO PIÙ SEMPLICE ADOPERANDO LA CLASSE NSXMLPARSER. SAREMO COSÌ IN GRADO DI RACCOGLIERE E MOSTRARE I CONTENUTI DEGLI RSS CONSUMANDO POCHISSIMA BANDA

L'evoluzione apportata dal web negli ultimi due decenni nel campo della comunicazione, ha reso possibile l'accesso a migliaia di fonti informative, eterogenee sia in termini di contenuti ma anche di presentazione delle informazioni. Verso la fine del millennio è maturata la necessità di definire un formato unico che consentisse l'interscambio di tali dati tra software e computer eterogenei ma che fosse anche facilmente leggibile ed analizzabile da un essere umano: da questa realtà ha preso forma l'XML.

COM'È FATTO UN DOCUMENTO XML

All'interno di un file XML prendono posto uno o più *tag*, entità testuali chiamate *elementi*, delimitate dal simbolo minore (<) e maggiore (>) che contengono le informazioni associate a tale marcatore. Ogni tag può essere del tipo *<tag>contenuto</tag>* oppure *<tag />*, può contenere al suo interno altri tag in numero teoricamente illimitato, e può inoltre avere dei valori associati, chiamati *attributi* che prendono posto subito dopo il nome del tag, prima del simbolo ">"; ecco un esempio in cui vengono adoperate tutte queste caratteristiche:

```
<?xml version="1.0"?>
<note>
<nota>
    <autore>Andrea</autore>
    <titolo>Ricordarsi di...</titolo>
    <corpo>Ricordarsi di inviare una mail.
                                    </corpo>
    <foto url="andrea.png" width="160"
                                height="220"/>

</nota>
<nota>
    <autore>Andrea</autore>
    <titolo>Comprare...</titolo>
    <corpo>Comprare cavo di rete 10-100-
                                    1000</corpo>
    <foto url="rete.png" width="320"
                                height="480" />

</nota>
<nota>
</nota>
</note>
```

La struttura di un file XML è a discrezione dello sviluppatore, anche per quanto riguarda i nomi dei tag. È possibile impostare delle regole che consentono di limitare il tipo, il numero e i valori contenuti nei suoi elementi, adoperando DTD o XML Schema. La descrizione qui fatta dell'XML è estremamente semplificata e le migliaia di risorse disponibili online comprese le decine di libri pubblicati rendono certamente maggiore giustizia a questo formato del W3C. Uno dei maggiori difetti che gli vengono attribuiti è la sua prolissità, che ha spinto molti sviluppatori ad optare per JSON (*JavaScript Object Notation*). Con il proliferare di siti e blog ha preso vita il formato ora conosciuto come *Really Simple Sindication*, RSS, inizialmente nato con il nome di RDF Site Summary negli uffici di Netscape. L'RSS (come il suo concorrente, Atom) non è nient'altro che un file XML strutturato con delle regole precise, il cui scopo è quello di organizzare e rendere fruibili all'esterno di un sito web le informazioni che vengono pubblicate al suo interno senza richiedere la consultazione ripetuta delle sue pagine. Questo formato è presente in milioni di siti, anche merito dell'utilizzo di piattaforme di blogging e CMS che ne hanno automatizzato la creazione, e l'icona con sfondo arancione e righe bianche è ormai divenuta un simbolo universalmente riconosciuto per identi-

CD WEB
iPhone_12.zip
cdrom.ioprogrammo.it

Fig. 1: La visualizzazione dei titoli dei feed

REQUISITI

Conoscenze richieste
Objectivive-C

Software
MacOS X 10.5.4 o superiore

Impegno

Tempo di realizzazione

ficarlo. Il pregio di tale formato risiede anche nel fatto che consente di ridurre il traffico dati dei siti, poiché un software che adopera RSS per verificare se i suoi contenuti sono stati aggiornati richiede un minore scambio di byte rispetto a quello necessario per effettuare il caricamento completo necessario per visualizzare interamente le pagine web (costituite da HTML, Javascript e immagini). A differenza dell'XML, in RSS alcuni campi sono obbligatori e nel prossimo paragrafo avremo modo di approfondire questo discorso. Un aggregatore, o RSS reader, è quel software che consente di gestire e consultare tali file RSS. In questo articolo e nel prossimo andremo a realizzare un RSS reader, adopereremo a tale scopo la classe *NSXMLParser*, presentando i titoli sequenzialmente in una tabella, e mostreremo i singoli contenuti in base alla selezione effettuata. *NSXMLParser* è un parser, una classe dedicata all'analisi di file XML, di tipo *SAX*, realizzato in Objective-C e disponibile all'interno dell'SDK.

IL FILE RSS

Ecco un esempio di come può essere strutturato un RSS versione 2.0:

```
<?xml version="1.0"?>
<rss version="2.0">
   <channel>

      <title></title>
      <link></link>
      <description></description>

      <language></language>
      <pubDate></pubDate>
      <lastBuildDate></lastBuildDate>
 <item>

         <title></title>
         <link></link>
         <description></description>
         <pubDate></pubDate>
</item>

</channel>
</rss>
```

All'interno del tag *rss* ne è contenuto un altro chiamato *channel*, che presenta prima una serie di campi che descrivono le caratteristiche generali del feed, come titolo, link ad una risorsa web, generalmente questo campo contiene il sito a cui è associato il file, trova poi posto la descrizione del file, la lingua, la data di pubblicazione, e successivamente si avranno uno o più tag *item* che rappresentano le singole notizie. Ogni *item* può contenere diversi campi, tra cui il titolo, il link alla pagine completa, la descrizione, la data di pubblicazione. Quelli mostrati sono sono alcuni dei tag disponibili, ma nessuno di questi è obbligatorio, a parte *title*, *link* e *description* per quanto riguarda la descrizione del feed, e *title* o *description* per il contenuto del singolo *item*.
Consultando le specifiche relative alla versione utilizzata è possibile realizzare un feed compliant con tale documentazione, avendo garanzia che sarà accettato dalla maggior parte dei feed reader.

SAX E DOM

Quando si adopera un parser per analizzare un file XML/RSS, esistono principalmente due approcci per accedere ai suoi contenuti: uno è quello basato su SAX, l'altro su DOM. Un parser che adopera SAX, acronimo di *Simple API for XML*, si basa su un certo numero di metodi (viene detto di tipo event-driven) che vengono invocati quando la libreria incontra determinati elementi del file XML, all'interno di tali metodi è quindi cura del programmatore gestire i contenuti incontrati e memorizzarli in una struttura dati che verrà poi mostrata all'utente; l'analisi del documento è quindi di tipo sequenziale, partendo dalla radice del file giungendo fino all'ultimo ramo.
L'approccio che adopera invece il DOM (*Document Object Model*, giunto alla versione chiamata "Level 3"), è del tipo tree-based, poichè analizza una copia completa della struttura del file XML, la cui struttura è appunto *ad albero*, mantenendola interamente in memoria. Un parser di tipo SAX viene definito *streaming parser* proprio per il fatto che attraversa e analizza la struttura in sequenza, e si rivela proprio per questa caratteristica come la soluzione migliore quando si analizzano file di dimensioni relativamente grandi; se il proprio obiettivo è quello di non occupare eccessiva memoria conviene optare per SAX, mentre si può scegliere indifferentemente per l'uno o l'altro approccio quando si parsano documenti di modeste dimensioni. Perchè allora non scegliere sempre una libreria che adopera SAX?
In alcune situazioni è necessario avere accesso all'intera struttura del file immediatamen-

te, come avviene per il DTD, adoperato per validare un documento, l'XSLT, utilizzato per applicare trasformazioni/conversioni di documenti XML, e XPath, utilizzato per accedere ai contenuti dell'XML, e in questi casi il DOM si presenta come la soluzione ideale, sempre a condizione di non avere strutture dati di dimensioni eccessive.
Per concludere, citiamo un ultimo pregio dell'approccio SAX: nel caso ci trovassimo nella situazione di dover prelevare un file XML di medie/grandi dimensioni da una sorgente remota (web o intranet) per ottenere una precisa informazione, grazie alla natura stessa di tale tipo di parser potremo interrompere il download della risorsa quando avremo raggiungo il tag desiderato, ciò comporterà un risparmio di traffico che in alcuni casi potrebbe essere rilevante. Con l'approccio DOM ciò non è possibile, poiché il file deve sempre essere trasferito e memorizzato completamente sul computer locale prima di effettuare qualunque analisi.

LIBRERIE A CONFRONTO

Nonostante in questo articolo andremo ad utilizzare solo la classe *NSXMLParser* è opportuno sapere che ne esiste un'altra, utilizzabile sia con SAX che DOM, realizzata in linguaggio C e chiamata *libXML2* fornita nell'SDK (nata in seno al progetto Gnome), il cui utilizzo però si rivela sicuramente meno intuitivo rispetto alla prima, ma le cui prestazioni sono nettamente superiori, spesso di un ordine di grandezza, merito ovviamente del linguaggio a più basso livello che è stato utilizzato. A fianco di queste due soluzioni ne trovano poi posto altre, disponibili nel web che si propongono come un'alternativa fornendo un approccio solamente tramite DOM:

- **TBXML**: libreria di tipo DOM con approccio leggero (in termini di risorse), realizzata in Objective-C, non effettua la validazione , non supporta XPath, e supporta solo la lettura del file XML, non la modifica.

- **TouchXML**: libreria di tipo DOM che mima l'approccio di *NSXMLParser*, realizzata in Objective-C, supporta XPath ma come *TBXML* è solo per lettura di file XML.

- **KissXML**: libreria di tipo DOM che mima l'approccio di *NSXMLParser*, realizzata in Objective-C, basata su *TouchXML*, alla quale aggiunge la possibilità di modificare l'XML.

- **TinyXML**: libreria realizzata in C di tipo DOM, supporta lettura e scrittura di file XML, ma non *XPath*, funzionalità implementabile adoperando un'altra libreria associata chiamata *TyinyXPath*.

- **GDataXML**: libreria di tipo DOM che mima l'approccio di *NSXMLParser*, realizzata in Objective-C, sviluppata da Google, supporta lettura e scrittura di file XML e *XPath*.

Per quanto riguarda la scelta tra le due librerie fornite all'interno dell'SDK, *NSXMLParser* e *libXML2*, questo dipende dalle dimensioni dei file XML coinvolti, ma anche dalla propria esperienza nel linguaggio C, obbligatorio quando si opta per la seconda libreria; nel caso di indecisione si possono testare le prestazioni delle due librerie sui propri file XML/RSS modificando un progetto fornito all'interno della documentazione presente sia nel sito Apple che allegata in XCode, il cui nome è *XMLPerformace*, raggiungibile nella sezione "Related sample code" della *NSXMLParser Class Reference*. Anche se *libXML2* è ineguagliabile per la velocità con cui analizza un documento resta sempre da considerare la necessità di adoperare il lin-

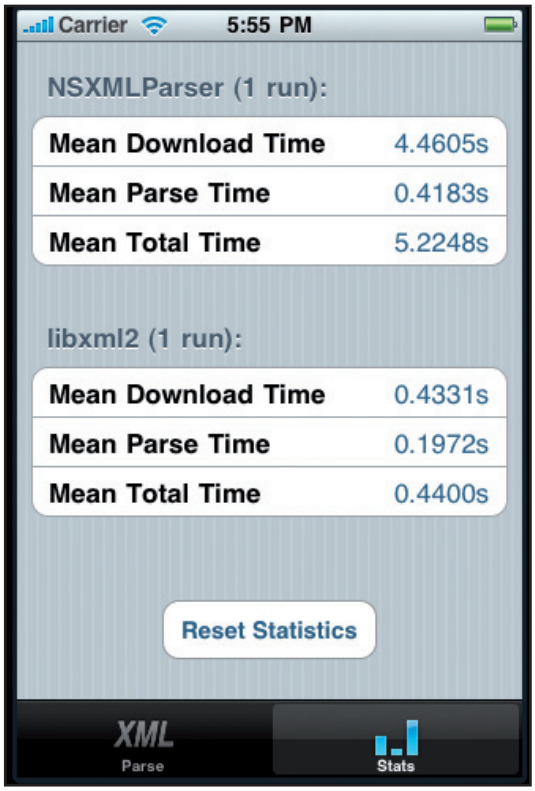

Fig. 2: Un test effettuato analizzando le top 300 songs di itunes mostra le differenti prestazioni delle due librerie fornite nell'SDK

NOTA

SPECIFICHE XML, RSS 1.X E 2.X

Per mantenersi aggiornati sulle specifiche riguardanti XML ed RSS, conviene fare affidamento sui siti ufficiali:

www.w3.org/XML/

http://web.resource.org/rss/1.0/spec

http://cyber.law.harvard.edu/rss/rss.html

guaggio C, che per alcuni utenti si rivela lo scoglio maggiore. Se invece si vuole adoperare DOM la scelta può cadere su qualunque delle cinque librerie precedentemente descritte se si vuole solo analizzare il documento XML, mentre il cerchio si restringe in caso si dovesse adoperare XPath oppure fosse necessario modificare il file. Da test empirici è risultato che TBXML si è rivelata una delle migliori in termini di velocità e di occupazione di memoria, ma per il fatto che molte librerie vengono modificate costantemente questa affermazione potrebbe già essere invalidata mentre leggete queste righe; comunque per documenti di dimensioni medio-grandi, quando la velocità di analisi del documento è di primaria importanza, il suggerimento è di adoperare *libxml2* (approccio SAX), mentre conviene optare per TBXML per un approccio DOM, il cui pregio in confronto a libxml2 (approccio DOM) è ovviamente la possibilità di adoperare l'Objective-C. Un'ultima nota che potrebbe interessare chi desidera analizzare anche pagine HTML con tali parser: con tutte queste librerie è possibile analizzare anche pagine web di tipo XHTML o HTML 5, poichè entrambi i formati sono *applicazioni* dell'XML.

LE RISORSE PER IPHONE

Creazione dell'account, per scaricare l'SDK e consultare la documentazione: http://developer.apple.com/iphone

IL PROGETTO

Creiamo un nuovo progetto, di tipo "Utility Application", chiamandolo *rssparser*, modifichiamo la *MainView* inserendo al suo interno un *UITableView* (impostiamo delegate e datasource al view controller *Mainviewcontroller.h* tramite interface builder), al quale accederemo attraverso un IBOutlet inserito all'interno del file *MainViewController.h*, che chiameremo *rssTable*, ricordandoci sempre di invocare il release all'interno del metodo *dealloc* per evitare leaks. Rimuoviamo il pulsante posizionato automaticamente in basso a destra con l'icona "*i*" perchè non è necessario in questo progetto.
Decommentiamo il metodo *viewDidLoad* all'interno di *MainviewController.c* e inseriamo il seguente codice:

```
NSURL *theURL = [NSURL URLWithString:@
                "http://rss.cnn.com/rss/edition.rss"];

NSXMLParser *parser = [[NSXMLParser alloc]
                    initWithContentsOfURL:theURL];
[parser setDelegate:self];
BOOL parseResult = [parser parse];

NSLog(@"Risultato del parsing della risorsa :
%@", parseResult?@"completato":@"errore");
```

Con queste poche righe di codice abbiamo già configurato ed avviato il parser. Nella prima riga abbiamo creato un oggetto di tipo *NSURL* contenente il link al file RSS, nella seconda è stata istanziata ed inizializzata una variabile di *NSXMLparser*, successivamente abbiamo impostato il suo *delegate* alla classe corrente in modo da ricevere tutti gli eventi che verranno generati dal parser durante la sua navigazione all'interno del file. Infine, invochiamo il metodo *parse* che dà inizio all'analisi del documento e mostriamo il risultato dell'evento nella console di debug tramite *NSLog*. Ora che abbiamo implementato il blocco di codice responsabile dell'avvio del parsing apriamo una parentesi sui metodi che il parser invierà alla nostra classe, che svolge il ruolo di *delegate*.

IL FUNZIONAMENTO DI NSXMLPARSER

Utilizziamo un file XML molto semplice, contenente come informazioni la versione, un articolo e una descrizione associata

```
<?xml version= "1.0" encoding="UTF8">
        <articolo autore="Andrea Leganza">
<titolo>Un feed reader per tutti</titolo>
<descrizione>Articolo dedicato ai file
                                XML.</descrizione>
        </articolo>
```

per mostrare come si comporta un'istanza della classe *NSXMLParser*, descrivendo la sequenza degli eventi inviati al suo delegate, che deve rispettare il protocollo richiesto chiamato *NSXMLParserDelegate*:

1. Avvio del parsing del documento;
2. Trovato inizio del tag *articolo* con attributo *autore* e valore "Andrea Leganza";
3. Trovato inizio del tag *titolo*;
4. Trovati caratteri all'interno del tag *titolo*: "Un feed reader per tutti.";
5. Trovata fine del tag *titolo*;
6. Trovato inizio del tag *descrizione*;
7. Trovati caratteri all'interno del tag *descrizione*: "Articolo dedicato ai file XML.";
8. Trovata fine del tag *descrizione*;
9. Trovata fine del tag *articolo*;
10. Fine del parsing del documento.

I passi *1* e *10* avvengono una sola volta duran-

te la fase di parsing, mentre quelli di inizio/fine elemento (*parser:didStartElement** e *parser:didEnd Element**) una volta per ogni tag incontrato; come si nota l'analisi procede andando sempre più in *profondità*, per poi *risalire* quando trova i vari tag di chiusura. Se avessimo inserito una sequenza di *n* tag *articolo*, avremo avuto *n* ripetizioni della sequenza dei passi da 2 a 9. Passiamo ora dalla versione con pseudocodice alla sequenza di chiamate dei metodi effettivamente richiesti al *delegate*:

```
1.  parserDidStartDocument:

1.  parser:didStartElement:namespaceURI:
    qualifiedName:attributes: //trovato tag articolo

1.  parser:didStartElement:namespaceURI:
    qualifiedName:attributes: //trovato tag titolo

2.  parser:foundCharacters:
    //trovato contenuto per tag titolo
3.  parser:didEndElement:namespaceURI:
    qualifiedName: //fine tag titolo

4.  parser:didStartElement:namespaceURI:
    qualifiedName:attributes: //trovato tag
    descrizione

5.  parser:foundCharacters:
    //trovato contenuto per tag descrizione

6.  parser:didEndElement:namespaceURI:
    qualifiedName: //fine tag descrizione

7.  parserDidEndDocument:
```

Oltre a questi ne esistono ovviamente molti altri, come quelli scatenati quando il parser incontra caratteri particolari, tipi di informazioni non prettamente testuali (*CDATA*), commenti, o il DTD; per avere una visione completa di tutti gli eventi che il delegate può ricevere basta consultare la documentazione relativa a *NSXMLParser Delegate*.
L'ultimo metodo che è sempre opportuno implementare in aggiunta ai precedenti è quello invocato dal parser per identificare e gestire le situazioni di errore, chiamato *parser:parseErrorOccurred:*. Un errore può presentarsi per diversi motivi, come tag non chiusi, oppure caratteri non accettati, in pratica in tutte quelle situazioni in cui il parser è indeciso sul da farsi perché ha incontrato una struttura non coerente. All'interno del metodo *parser:parse ErrorOccurred* si può decidere come procedere dopo aver interrogato l'istanza di NSError chiamata *parseError* i cui codici di errore sono elencati in una legenda presente nella sezione *Parser Error Constants* della documentazione associata alla classe *NSXMLParser*.

IL NUMERO DI INVOCAZIONI

La sequenza di eventi descritti nel precedente paragrafo per il file XML mostrato è speculare a quella per un file RSS ovviamente con alcune differenze in termini di numeri: in questo caso il numero di invocazioni di *parser:didStart Element:...* (e del complementare *parser:didEndElement:...*) è nettamente superiore: otterremo infatti una sequenza di eventi dovuta alla parte di descrizione del feed, e successivamente, quando il parser giungerà alle notizie, identificate dai tag *item*, in numero pari a molte decine, anche centinaia.
Per valutare il numero complessivo di invocazioni di metodi richieste al delegate senza avviare il software, basandosi solo sulla struttura del file RSS, basterà calcolare il risultato della seguente formula: $S+(K*2)+(n*2)+(n*(j*2))+d$ dove

- ***S***: costante pari a 2, i due metodi di inizio e fine documento
- ***K***: costante, numero di tag utilizzati per la descrizione del feed;
- ***n***: numero di notizie (tag *item*)
  j: numero di tag presenti all'interno della singola notizia (tag item);
- ***d***: numero di invocazioni di *foundCharacters*:

È ovviamente possibile calcolare tutti questi parametri inserendo una variabile che contabilizzi il numero di invocazioni di ogni metodo quando si avvia un parsing.
Questi calcoli possono risultare utili quando si è al contempo consumatore (tramite il software iPhone dell'RSS), ma anche responsabile della sua generazione. In tal modo si possono apportare modifiche strutturali per velocizzarne l'analisi, riducendo accapo e caratteri inutili, oppure semplificando la struttura accorpando tag ed adoperando attributi.
Nel prossimo articolo tratteremo approfonditamente del prelievo delle informazioni all'interno dei tag e mostreremo la loro descrizione con le informazioni associate. Buona programmazione.

Andrea Leganza

L'AUTORE

Andrea Leganza
Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali, e non, su piattaforme e con linguaggi diversi. Certificato Adobe ACE - Adobe Flex 3 and AIR Certified Expert, e EUCIP Core, appassionato di fotografia, lingua giapponese e istruttore di nuoto FIN, è attualmente impegnato in numerosi progetti multimediali, anche con iPhone, con alcune società nazionali ed internazionali; è contattabile su neogene@tin.it o direttamente sul sito www.leganza.it.

UN RSS READER PER IPHONE

ADOPERANDO LA CLASSE NSXMLPARSER, POSSIAMO INTERAGIRE CON I CONTENUTI DI UN FEED RSS. IN QUESTO ARTICOLO MOSTREREMO COME FARE E COME IMPLEMENTARE UNA STRUTTURA DATI PER LA VISUALIZZAZIONE DELLE INFO RICEVUTE

CD WEB
iPhone_RSS.rar
cdrom.ioprogrammo.it

Nell'articolo precedente abbiamo trattato in maniera teorica del parsing di feed RSS, descrivendo a grandi linee il formato RSS, ciò ha permesso di mostrare come questo sia semplicemente un file XML con una precisa struttura; in un secondo momento sono stati descritti i due tipi di parser forniti nell'SDK, DOM e SAX, dei quali abbiamo spiegato pregi e difetti; successivamente siamo entrati nel vivo del corso mostrando il funzionamento del parser *NSXMLParser*, il quale invoca sequenzialmente alcuni metodi che devono essere implementati dal suo delegate, comportamento che viene definito *event-driven*. In questa ultima parte mostreremo il contenuto dei metodi del *delegate* e ne spiegheremo il funzionamento.

IL PARSING DEL FILE RSS

Riproponiamo per comodità il metodo *viewDidLoad* nel quale diamo inizio all'operazione di parsing:

```
- (void)viewDidLoad {

    //Allochiamo ed inizializziamo l'array che conterrà le singole news;
    notizie = [[NSMutableArray alloc] init];

NSURL *theURL = [NSURL URLWithString:@"http://rss.cnn.com/rss/edition.rss"];
NSXMLParser *parser = [[NSXMLParser alloc] initWithContentsOfURL:theURL];
[parser setDelegate:self];
BOOL parseResult = [parser parse];
NSLog(@"Risultato del parsing della risorsa : %@", parseResult?@"completato":@"errore");
}
```

Spieghiamo brevemente il codice appena proposto: abbiamo creato un'istanza di *NSURL* nella quale è stato inserito l'URL del feed che vogliamo analizzare, successivamente istanziamo un oggetto della classe *NSXMLParser* impostandone come *delegate* la classe in cui questo codice è presente; infine, diamo inizio all'operazione di parsing. Ora che abbiamo dato inizio all'operazione di parsing sappiamo, dall'articolo precedente, che avremo una sequenza di chiamate ai seguenti metodi:

parserDidStartDocument: (1 chiamata)
parser:didStartElement:namespaceURI:qualifiedName: attributes: (n chiamate)
parser:foundCharacters: (m chiamate)
parser:didEndElement:namespaceURI:qualifiedName: (n chiamate)
parserDidEndDocument: (1 chiamata)

ParserDidStartDocument e *parserDidEndDocument* indicano l'inizio e il termine della procedura di parsing e generalmente sono adoperati per inizializzare e deallocare se necessario, tutte quelle variabili e strutture dati che verranno utilizzate durante la fase di analisi del documento RSS. *parser:didStartElement:namespaceURI:qualifiedName:attributes:* e *parser:didEndElement:namespaceURI:qualifiedName:* svolgono invece un ruolo molto importante, infatti ci informano quando è stato trovato un tag, aperto (<*tag*>) o chiuso (</*tag*>). *parser:foundCharacters:* viene invocato quando trova del testo tra un tag di apertura e uno di chiusura (<tag>testo</tag>). *parser:foundCharacters* viene invocato quando il parser incontra del testo all'interno di un tag. All'interno di questi metodi dovremo popolare una struttura dati apposita con le informazioni ricevute per ogni tag, che andremo poi a inserire all'interno di un array: al termine del parsing troveremo in tale array per ogni suo record tutti i dati associati al singolo tag incontrato. Generalmente l'operazione di parsing necessita delle seguenti strutture dati:

- una istanza della classe NSMutableString, *currentElement*, per condividere tra le varie chiamate dei metodi il tag che stiamo analizzando;
- n istanze della classe NSMutableString; nel nostro caso sono quattro, *currentFitle*, *currentDate*, *cur-*

REQUISITI

Conoscenze richieste
Objective-C

Software
Mac OS 10.5.x o superiore (10.6.x per l'SDK 3.x)

Impegno

Tempo di realizzazione

rentSummary e *currentLink*, ognuna conterrà il testo associato al relativo tag;

- una istanza del tipo NSMutableDictionary, con il nome *notizia*, che conterrà tutte le informazioni che verranno trovate per una singola news e associate alle *NSMutableString* (*titolo*, *data*, *sommario*, *link*).
- un array del tipo *NSMutableArray*, *notizie*, nel quale andremo ad inserire al termine del parsing di un tag di tipo item l'*NSMutableDictionary* appena popolato; tale istanza quindi alla fine del parsing conterrà n campi, uno per ogni news prensente nell'RSS.

```
NSMutableArray *notizie;

NSMutableDictionary *notizia;

NSMutableString * currentElement;

NSMutableString * currentTitle;
NSMutableString * currentDate;
NSMutableString * currentSummary;
NSMutableString * currentLink;
```

Perché adoperare istanze di *NSMutableString* invece di *NSString*? La flessibilità di *NSMutableString* ci consente di non dover allocare ed inizializzare ogni volta che troviamo un nuovo *titolo*, o *data*, o *sommario* o *link*, una nuova variabile, possiamo infatti rimuovere la stringa che tali variabili contengono semplicemente impostando il loro testo a @"", questa operazione avverrà quando avremo raggiunto il tag di chiusura e stiamo per passare ad analizzarne un altro. È importante ricordare che le istanze di *NSString* sono costanti e non abbiamo controllo su quando verranno deallocate, poichè durante il parsing ne verrebbero generate centinaia, non abbiamo modo di sapere se e quando queste verranno rimosse dalla memoria, situazione invece non presente con *NSMutableString*, classe che ci consente pieno controllo sul suo tempo di vita grazie all'invocazione di release. Prima di iniziare con metodi che si rivelano sicuramente più interessanti è importante porre l'attenzione al caso in cui per qualche motivo il parser fallisca l'operazione di analisi, i motivi possono essere molteplici, uno dei più frequenti può presentarsi quando non c'è sufficiente copertura sulla rete GSM, oppure non è presente una la rete WIFI, situazioni quindi che rendono impossibile connettersi al server in fase di richiesta della risorsa oppure in ricezione del file RSS se la connessione si interrompe; qualunque sia il tipo di errore questo interromperà il parsing, e nel caso in cui sia stato implementato il metodo *parseErrorOccurred*: sarà responsabilità del parser provvede ad invocarlo. È sempre consigliabile implementare tale metodo, presentando con un *UIAlerView* la causa dell'errore, informazione disponibile con un codice numerico all'interno del parametro *parseError*, in questo modo sarà anche più semplice ricevere segnalazioni da parte degli utenti in caso di un non corretto funzionamento di questo componente. Per un elenco completo dei quasi 100 codici di errore possibili è necessario consultare la documentazione associata alla classe *NSXMLParser*, nella sezione chiamata *Parser Error Constants*.

```
- (void)parser:(NSXMLParser *)parser
                parseErrorOccurred:(NSError *)parseError {

NSLog(@"Errore di parsing: %f", [parseError code]);
UIAlertView * errorAlertView = [[UIAlertView alloc]
   initWithTitle:@"Errore durante l'analisi "
   message:[NSString stringWithFormat:@"Errore di
                                parsing: %f",[parseError code]]
   delegate:nil
   cancelButtonTitle:@"ok"
   otherButtonTitles:nil];
   [errorAlertView show];
   [errorAlertView release];
}
```

Ora bisogna ricordare quale sia la struttura più comune di un file RSS:

```
<?xml version="1.0"?>
<rss version="2.0">
   <channel>
      <title></title>
      <link></link>
      <description></description>
      <language></language>
      <pubDate></pubDate>
      <lastBuildDate></lastBuildDate>
 <item>
         <title></title>
         <link></link>
         <description></description>
         <pubDate></pubDate>
   ...//possibili altri parametri
</item>
...//ripetizioni di <item>...</item>
</channel>
</rss>
```

Inizializziamo all'interno del metodo *parserDidStartDocument* prima la stringa che conterrà il nome del tag che il parser ha incontrato, e facciamo lo stesso con le stringhe che conterranno i testi trovati all'interno dei vari tag di ogni item; i tag che ci interessano sono il titolo, la data, il sommario e il link alla pagina web: *title*, *pubDate*, *description* e *link*:

Fig. 1: Il risultato finale di questo progetto

```
- (void)parserDidStartDocument:(NSXMLParser *)
                                                parser{
    NSLog(@"Parsing del documento iniziata.");
    currentElement = [[NSMutableString alloc] init];
    currentTitle = [[NSMutableString alloc] init];
    currentDate = [[NSMutableString alloc] init];
    currentSummary = [[NSMutableString alloc] init];
    currentLink = [[NSMutableString alloc] init];
}
```

Deallocheremo tali variabili solo alla fine del parsing, all'interno di *parserDidEndDocument*: e provvederemo infine ad aggiornare la tabella che adoperiamo per mostrare i titoli dei feed RSS.

```
- (void)parserDidEndDocument:(NSXMLParser *)parser{
    [currentElement release];
    [currentTitle release];
    [currentDate release];
    [currentSummary release];
    [currentLink release];
    NSLog(@"Fine del parsing.");
    NSLog(@"Notizie contiene %d elementi", [notizie
                                                count]);
    //Refresh dei contenuti della UITableView
    [rssTable reloadData];
}
```

Dopo aver inizializzato la maggior parte delle variabili necessarie per il parsing, entriamo nel vivo del codice, analizzando *parser: didStartElement: namespaceURI: qualifiedName: attributes*:

```
- (void)parser:(NSXMLParser *)parser
                    didStartElement:(NSString *)elementNa
                   me namespaceURI:(NSString *)namespa
                   ceURI qualifiedName:(NSString *)qName
                    attributes:(NSDictionary *)attributeDict{
    [currentElement setString:[elementName copy]];
    if ([currentElement isEqualToString:@"item"]) {

      notizia = [[NSMutableDictionary alloc] init];
    }

}
```

Appena il parser ci notifica che ha trovato un tag di apertura, accessibile interrogando il parametro *elementName*, lo memorizziamo all'interno di *currentElement*; come avevamo mostrato prima, la struttura di un file RSS presenta molti tag diversi, *description*, *language*, *pubDate*, *lastBuildDate* e tanti altri, a noi interessano solo i tag <*item*>, con i tag "figli" <*title*>, <*link*>, <*description*>, <*pubDate*>, che contengono le singole notizie, e per tale motivo preleveremo i dati dall'RSS solo quando ci troveremo all'interno di tali marcatori. Ecco svelato l'utilità della variabile *currentElement*, questa ci servirà nel metodo *parser: foundCharacters* per popolare le nostre *NSMutableString* e l'*NSDictionary* solo quando siamo all'interno di tali tag, infatti *parser: foundCharacters* non fornisce alcun parametro che indichi in che tag siamo spetta quindi a noi tenerne traccia; adoperiamo quindi un semplice confronto tra stringhe *[elementName isEqualToString:@"tagname"]* per capire se il tag corrente è di interesse o no. Nel caso in cui siamo all'interno del tag <*item*> provvediamo a inizializzare un *NSMutableDictionary* al cui interno inseriremo i contenuti di *title*, *link*, *description* e *pubDate*. Ad una prima invocazione *didStartElement*: fornirà come *elementName* *@"title"*, alla seconda invocazione *@"title"*, poi *@"description"* ed infine *@"pubDate"*. Un'ultima nota riguarda il parametro *attributeDict*, questo è un *NSDictionary* che contiene tutti gli attributi trovati per un tag, ad esempio, se il tag che abbiamo incontrato è del tipo <*tag name="Andrea" second="Leganza" age="31"*>*Informatic Engineerer*</*tag*> invocando su tale variabile il metodo *objectForKey*: adoperando come chiave il nome degli attributi presenti otterremo il valore di tali proprietà.

```
[attributeDict objectForKey:@"name"]; restituisce la
                                          stringa @"Andrea"
[attributeDict objectForKey:@"second"]; restiuisce la
                                          stringa @"Leganza"
[attributeDict objectForKey:@"age"]; restiuisce la
                                          stringa @"31"
```

Adesso che siamo all'"interno" di un tag e possono presentarsi due situazioni: è presente un ulteriore tag di apertura, come avviene per <*item*>, con i suoi tag "figli", nel qual caso verrà invocata un'altra volta *didStartElement*:, oppure, come avviene quando siamo in uno qualsiasi dei tag "figli", verrà invocato parser: *foundCharacters*: ad indicare che abbiamo incontrato del testo all'interno del marcatore, del tipo <*tag*>*testo_contenuto*</*tag*>:

```
- (void)parser:(NSXMLParser *)parser
                        foundCharacters:(NSString *)string{
    if ([currentElement isEqualToString:@"title"]) {
      [currentTitle appendString:string];
    } else if ([currentElement isEqualToString:@"link"]) {
      [currentLink appendString:string];
    } else if ([currentElement isEqualToString:@"descripti
                                                  on"]) {
      [currentSummary appendString:string];
    } else if ([currentElement
                              isEqualToString:@"pubDate"]) {
      [currentDate appendString:string];
    }
}
```

Come abbiamo appena detto, *currentElement* ci permette di identificare il tag in cui ci troviamo,

NOTA

RIFERIMENTI WEB

Creazione dell'account, per scaricare l'SDK e consultare la documentazione previa registrazione gratuita: http://developer.apple.com/iphone/

all'interno di *parser:foundCharacters:* popoliamo la *NSMutableString* in base al tag (*title,link,description,pubdate*) che è al momento analizzato. Quando raggiungiamo il tag di chiusura di ogni singolo tag avremo un'invocazione di *didEndElement:*, solo nel caso in cui abbiamo raggiunto la fine della news, identifcato dal tag di chiusura *</item>* provvediamo a popolare con le *NSMutableString* generate nel metodo *foundCharacters* l'*NSMutableDictionary*, inizializzato nel metodo *didStartElement*. Concludiamo rimuovendo l'*NSMutableDictionary* e rimuovendo il contenuto delle *NSMutableString*, così saranno pronte per essere di nuovo popolate per il prossimo item. Ricordiamo che nel caso in cui il tag svolgesse il ruolo di mero contenitore di altri tag, come avviene per *<item>*, non avremo alcuna chiamata a *foundCharacters*, quindi ipotizzando che questo tag contenga solo i 4 tag "figli" mostrati, avremo 4 invocazioni a *foundCharacters* e non 5 come alcuni potrebbero erroneamente pensare.

```
- (void)parser:(NSXMLParser *)
            parser didEndElement:(NSString *)element
      Name namespaceURI:(NSString *)namespaceURI
                qualifiedName:(NSString *)qName{
if ([elementName isEqualToString:@"item"]) {
   [notizia setObject:[currentTitle copy] forKey:@"title"];
   [notizia setObject:[currentLink copy] forKey:@"link"];
   [notizia setObject:[currentSummary copy]
                                    forKey:@"summary"];
   [notizia setObject:[currentDate copy]
                                    forKey:@"date"];
   [notizia addObject:[notizia copy]];
   [notizia release];
   [currentTitle setString:@""];
   [currentDate setString:@""];
   [currentSummary setString:@""];
   [currentLink setString:@""];
   }
}
```

LA POPOLAZIONE DELLA TABELLA

Ora che abbiamo finalmente ottenuto le informazioni che desideriamo le andremo a mostrare in una *UITableView*; provvediamo prima a configurare la tabella:

```
- (NSInteger)numberOfSectionsInTableView:
                              (UITableView *)tableView {
   return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
        numberOfRowsInSection:(NSInteger)section {
   return [notizie count];
}
```

popoliamo quindi il corpo del metodo responsabile della visualizzazione nelle singole celle/righe, *tableView:cellForRowAtIndexPath*:

```
- (UITableViewCell *)tableView:(UITableView *)table
  View cellForRowAtIndexPath:(NSIndexPath *)indexPath
                                                {
static NSString *MyIdentifier = @"MyIdentifier";
UITableViewCell *cell = [tableView dequeueReusableCell
                                  WithIdentifier:MyIdentifier];
if (cell == nil) {
cell = [[[UITableViewCell alloc] initWithStyle:UITableVie
                wCellStyleSubtitle reuseIdentifier:MyIdentifier]
                                                autorelease];
   cell.imageView.image = [UIImage
                                  imageNamed:@"news.jpg"];
   cell.textLabel.adjustsFontSizeToFitWidth = YES;
}
int notiziaIndex = [indexPath indexAtPosition: [index
Path length] - 1];
cell.textLabel.text = [[notizie objectAtIndex: notizia
                                  Index] objectForKey: @"title"];
return cell;
}
```

A parte le tipiche operazioni di configurazione, abbiamo invocato il metodo *adjustsFontSizeToFitWidth* sulla *textLabel* della cella allo scopo di impostare l'autoresizing del testo, in modo da consentire una completa visualizzazione del titolo della news, evitando l'autotroncamento del testo, operazione effettuata

Fig. 2: Selezionando una cella usciremo dall'applicazione e apriremo il link con Safari

per default da tale *UILabel*; otteniamo l'indice della notizia per la relativa cella e lo memorizziamo all'interno della variabile *notiziaIndex*, andiamo quindi immediatamente a utilizzare nella riga successiva tale variabile per ottenere il relativo oggetto contenuto nell'array *notizie*, e preleviamo il titolo della news utilizzando la chiave *title* dell'*NSDictionary* selezionato.

LA VISUALIZZAZIONE DELLE PAGINE

Ora che abbiamo presentato correttamente a video i titoli, possiamo decidere di conseguenza il modo più adatto con cui rispondere alla selezione dell'utente; avremmo potuto utilizzare una *UITextView* o un *UIWebView* ma approfittiamo di questo contesto per esporre una nuova funzionalità fornita dall'SDK: l'apertura di link web utilizzando Safari. In uno dei primi articoli dedicati alla programmazione su iPhone abbiamo mostrato come realizzare un mini web browser adoperando la classe *UIWebView*, il cui cuore è lo stesso di Safari, l'open source WebKit, ma è possibile anche aprire link senza dover utilizzare tale classe, a patto di accettare l'inconveniente di dover terminare la propria applicazione. Ogni applicativo espone al suo interno un'istanza condivisa (in gergo viene chiamata un *singleton*) appartenente alla classe *UIApplication*, ottenibile invocando su tale classe il metodo *sharedApplication*, tale istanza fornisce una quindicina di metodi, due dei quali sono quelli che più ci interessano in questo articolo: *canOpenURL*: e *openURL*:.

Il primo metodo restituisce come risultato un booleano (YES/NO) a seconda se il link che gli inviamo è associato ad almeno un software installato nel nostro telefono (esiste quindi almeno un software in grado di gestirne i contenuti), mentre il secondo provvede ad aprire tale link con il programma associato. Safari, il web browser installato di default nell' iPhone/iPod Touch, è impostato per default come software responsabile dell'apertura dei link del tipo *http://*, *https://* e *mailto*:, quindi, quando andremo a chiamare *canOpenURL* passando il link associato alla news, otterremo una risposta affermativa, potremo invocare *openURL* con la sicurezza che l'utente visualizzerà la pagina della notizia associata all'interno di Safari. Immediatamente dopo l'invocazione di tale metodo assisteremo alla chiusura del nostro software e all'apertura di Safari. Questa pratica si rivela utile nei casi in cui non si vuole impegnare tempo nella costruzione di un browser web interno, ma ha il grande limite di terminare l'utilizzo del software, obbligando l'utente, nel caso lo desiderasse, di dover riaprire il software e rieffettuare tutte le operazioni necessarie per mostrare nuovamente le news: è quindi una soluzione da adoperare solo nelle primissime fasi di sviluppo del software, successivamente è sempre opportuno realizzare un browser interno, operazione che impiega in genere poche decine di minuti.

```
- (void)tableView:(UITableView *)tableView didSelect
                    RowAtIndexPath:(NSIndexPath *)indexPath {
int notiziaIndex = [indexPath indexAtPosition: [index
                                        Path length] - 1];
//Il link viene prelevato adoperando la chiave link
                                        sull'NSDictionary.
NSString *notiziaLink = [[notizie objectAtIndex: notizia
                                        Index] objectForKey: @"link"];

//Per sicurezza vengono sostituiti, se presenti, caratteri
    non compatibili con il formato accettatto per i link con
    opportune sequenze come ad esempio %20 per rappre
                                        sentare lo spazio
notiziaLink = [notiziaLink stringByAddingPercentEscapes
                                        UsingEncoding:NSUTF8StringEncoding];
//Nel caso fossero presenti simboli di ritorno a capo
                                        vengono rimossi
...
```

sharedApplication consente non solo di aprire link a siti web, ma anche effettuare telefonate, inviare SMS, o anche inviare email, previa sempre conferma da parte dell'utente:

```
[[UIApplication sharedApplication] openURL:[NSURL URL
                                        WithString:@"tel://123456789"]]
[[UIApplication sharedApplication] openURL:[NSURL URL
                                        WithString:@"sms://123456789"]]
[[UIApplication sharedApplication] openURL:[NSURL URL
WithString:@"mailto:emailAdress?subject=helloSubject&
                body=hello my friend has been a while..."]];
```

La necessità di richiedere conferma quando si effettua una chiamata o si invia un SMS è per evitare abusi da parte di sviluppatori non proprio ortodossi, che potrebbero effettuare chiamate senza autorizzazione (potendo quindi ascoltare le conversazioni altrui), con conseguente spesa da parte dell'utente, oppure inviare SMS con dati sensibili senza che l'utente ne abbia controllo; ovviamente questi due metodi sono validi solo su iPhone e in questo caso se invocassimo il metodo *canOpenURL* su un iPod Touch otterremo risposta negativa.

L'ultimo metodo, quello di invio email era una delle poche soluzioni disponibili prima del rilascio dell'SDK 3, ora grazie all'introduzione del componente *MFMailComposeViewController* è sempre meno adoperato, e si possono applicare le stesse considerazioni fatte per l'apertura di link http/s con Safari. È anche possibile lanciare l'applicativo google maps installato nel dispositivo http://maps.google.com/<*parametri*>), e una pagina dell'App Store (http://phobos.apple.com/<*parametri*>). È possibile inoltre registrare la propria applicazione come responsabile dell'apertura di uno o più formati di link ma è adoperato solo in rari casi.

Andrea Leganza

Andrea Leganza Laureato in Ingegneria Informatica, certificato Adobe ACE - Adobe Flex 3 and AIR Certified Expert, EUCIP Core, Sun Certified Programmer for JAVA 6, istruttore di nuoto FIN di 2° Livello, è attualmente impegnato in numerosi progetti multimediali, anche con iPhone e iPad, con alcune società nazionali ed internazionali, è contattabile su neogene@tin.it o su www.leganza.it.

Questo approfondimento tematico è pensato per chi vuol imparare a programmare e creare software per l'Apple iPhone.
La prima parte del testo guida il lettore alla conoscenza degli strumenti necessari per sviluppare sulla piattaforma mobile di Cupertino. Le sezioni successive sono pensate per un apprendimento pratico basato su esempi di progetto: la creazione di un browser su misura, la gestione dell'interfaccia, la programmazione di un'agenda e di una to do list, la gestione corretta di celle e tabelle, l'utilizzo dell'accelerometro, la progettazione di un RSS reader e via dicendo. Una serie di esempi pratici da seguire passo passo che – creando applicazioni testabili e perfettamente funzionanti - spingono il lettore a sperimentare sul campo il proprio livello di apprendimento e lo invitano a imparare divertendosi.

PuntoInformatico
Internet dal 1996
PuntoInformatico | Libri
www.punto-informatico.it